DOCUMENT RESUME

ED 057 580 EM 009 420

AUTHOR	Grant, Richard; And Others
TITLE	LOGO Teaching Sequences on Numbers and Functions and Equations. Teacher's Text and Problems.
INSTITUTION	Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	R-2165
PUB DATE	30 Jun 71
NOTE	230p.; Programming-Languages as a Conceptual Framework for Teaching Mathematics, Volume Two; See also EM 009 419, EM 009 421, EM 009 422
EDRS PRICE	MF-$0.65 HC-$9.87
DESCRIPTORS	*Computer Assisted Instruction; *Mathematics Instruction; Numbers; *Programing Languages; Set Theory; *Teaching Guides
IDENTIFIERS	Project LOGO

ABSTRACT

The teacher's texts for two teaching sequences in the LOGO mathematics course are presented in this second volume of a four-volume report. The material presented here is designed to be a broad overview of the application of LOGO to the topics of numbers and functions. A variety of alternative paths and approaches are presented; in each case the emphasis is on crucial points and on possible pitfalls and difficulties. The sequence on numbers is not meant to accompany a first exposure to the subject, but rather, a careful retracing of steps. The level of presentation in this unit is extremely detailed, and the reader is encouraged, on first reading, to skip around as his interests dictate. The sequence on functions is written more freely. The idea of function as a black-box is here concretely realized as are many other aspects of the set-theoretic approach to functions which otherwise trouble students by their "vagueness." For Volumes I, III, and IV of the report see EM 009 419, EM 009 421, and EM 009 422. (Author/JY)

ED057580

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 2165 Volume 2

PROGRAMMING-LANGUAGES AS A CONCEPTUAL FRAMEWORK FOR TEACHING MATHEMATICS

LOGO Teaching Sequences on

Numbers

and

Functions and Equations

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY.

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, N.W.
Washington, D. C. 20550

Contract NSF-C 615

30 June 1971

EM 009 420

ED057580

FOREWORD

The use of LOGO in the classroom requires that the teacher have a broad overview of the application of LOGO to the topics being treated. This volume contains materials intended to present such an overview in the topics of numbers and of functions. A variety of alternative paths and approaches are presented, in each case emphasis being placed on crucial points and on possible pitfalls and difficulties. The idea is not to present material in precisely the manner in which it is to be taught, but on giving the reader enough insight so that he can freely apply LOGO to the mathematical issues of direct concern to him. It may well be, for example, that only carefully selected portions are dealt with in the classroom, or that, a completely different approach is taken, arising perhaps from a suggestion in the text or the set of problems appended to each unit.

The material on numbers is not meant to accompany a first exposure to the subject, but rather, a careful retracing of steps. Not having to worry about what numbers are, the student can contrast the various sorts of number representations, concerning himself at each stage with understanding of the basic algorithms required. Writing these algorithms as LOGO procedures enables him to define them more precisely and concretely than otherwise, and gives him powerful means for using and extending them. The level of presentation in this unit is extremely detailed as befits the nature of the material and the reader is encouraged, on first reading, to skip around as his interests dictate.

The material on functions is much more suggestive and written more freely. The idea of function as black-box is here concretely realized as are many other aspects of the set-theoretic approach to functions which otherwise trouble students by their "vagueness".

Volume 2, Part 1

LOGO NUMBER UNIT

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

Richard Grant

Philip Faflick

Wallace Feurzeig

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

		Page
1.	Introduction to This Unit	1
2.	Mark Numbers	4
	2.1 Introduction to Mark Numbers	4
	2.2 Addition of Mark Numbers	5
	2.3 Comparing Mark Numbers	6
	2.4 Subtraction of Mark Numbers	9
3.	Compact Numbers	12
	3.1 Grouping Mark Numbers	12
	3.2 Conversion of Mark Numbers to Grouped Numbers	16
	3.3 A Compact Representation of Grouped Numbers	22
	3.4 Conversion of Compact Numbers to Mark Numbers	24
	3.5 Disordered Compact Numbers	27
	3.6 Standard Form	34
	3.7 Comparing Compact Numbers	37
	3.8 Subtraction of Compact Numbers	39
4.	More About Mark Numbers	44
	4.1 Multiplication of Mark Numbers	44
	4.2 Counting, Copying, and Compact Multiplication	49
	4.3 Division of Mark Numbers	52
5.	Place Numbers	57
	5.1 Positional Notation	57
	5.2 Addition of Place Numbers	60
	5.3 The Place Number Successor	64
	5.4 Converting Place Numbers to Standard Form	66
	5.5 Comparing Place Numbers	70
	5.6 Subtraction of Place Numbers	74
6.	Modern Numbers	80
	6.1 Digits	80
	6.2 Conversion Between Mod Numbers and Place Numbers	82

CONTENTS (continued)

Page

6.3 Mod Arithmetic Along the Lines of Place Arithmetic 84
6.4 Adding Money Numbers 89
6.5 Mod Number Addition 95
6.6 Mod Number Subtraction 96
6.7 Multiplication of Mod Numbers 100
6.8 Division of Mod Numbers 104

Appendices

Problems

LOGO Number Unit

Teacher's Text

1. Introduction to This Unit

The subject of this part of the LOGO mathematics course is numbers and their representation. It introduces several distinct ways of representing numbers, interprets the familiar operations of arithmetic in terms of each notational system, and develops the corresponding algorithms in the form of LOGO programs.

Four main representations are considered. We begin with a simple notation called mark numbers, similar to the marks used by children for scoring games. We extend this to a more compact notational system similar to Roman numerals, called compact numbers. We then introduce the idea of place value which makes possible a relatively pure form of positional notation, which we call place numbers. We evolve from this the relatively elaborate and efficient notation we know from familiar, everyday, current use as our numbers (these are called modern numbers, or mod numbers for short). Along the way we consider some related representations, such as "money numbers".

With each of these notations we develop LOGO procedures for performing the operations of arithmetic -- that is, for counting (succession), adding, comparing, subtracting, multiplying, and dividing -- with the corresponding numbers. We also write procedures for converting numbers across these various representations. Often several distinct algorithms for performing an operation (such as comparing mark numbers) are developed.

Throughout the text we have generally adopted the conventional use of "number" for what should properly be called "numeral".

In those instances where it is important to make this distinction, we use the term "representation" or "notation" (for example, as in "numbers expressed in this representation").

While this unit is a new presentation of number arithmetic, it is, at the same time, a unit on programming as a constructive means of expressing mathematical procedures, and this is an integral part of the presentation.

The text is written for use primarily by teachers rather than students, though the material is certainly appropriate for use by capable and advanced students as a basis for independent study and work. To help teachers in preparing their classroom discussions, we have included a great deal of simple expository material in the text.

The unit can be properly regarded as a re-introduction to arithmetic rather than as the first introduction. (We have distinctly different ideas about the first introduction of concepts like number and algorithm than those used here.) The material is intended for use at about seventh grade level though we think it would be equally suitable with many fifth grade classes. Like any sound presentation of good mathematics it is also appropriate, though in a somewhat different way, for teaching older and more sophisticated students and teachers.

This unit illustrates one approach to the use of LOGO as a *conceptual framework* for teaching mathematics. It might appear feasible to teach the same material without using a computer, and indeed this is possible in principle. But, it would be

very difficult to do so in an effective way. An essential aspect of the presentation would be lost: we would no longer be providing the student with an active operational universe for constructing and controlling a mathematical process. This modified presentation would seriously impoverish the character and quality of the student's experience and of its educational benefit.

The main author of the unit is Richard Grant who advanced the central scheme, suggested the overall series of topics and their sequencing, and wrote the material on mark numbers and compact numbers. Philip Faflick wrote the chapters on place numbers and modern numbers. Wallace Feurzeig made occasional contributions to the ideas and writing, and edited the manuscript.

2. Mark Numbers

2.1 Introduction to Mark Numbers

Mark numbers are probably similar to the first way of expressing numbers man invented. We can imagine an ancient shepherd with a bag of pebbles, one pebble for every sheep. When he wants to count the sheep, he has them walk past in single file and removes one pebble from the bag as each sheep passes. If the bag is emptied just as the last sheep goes through, he knows that they are all there.

The bag of pebbles represents a number just as a word composed of X's does. They are both mark numbers; in the first pebbles are used as the marks, while in the second X's are used. Notches on the gun of a western gunfighter are another example.

An important virtue of mark numbers, besides their simplicity, is the ease with which one mark number can be changed into the next mark number. For example, by adding a pebble, an X, or a notch. This makes them very handy for keeping tallies; it never becomes necessary to erase the previous number in order to create the next. (The familiar modern form of mark numbers, which people often use in scoring games, for example 𝍸 𝍸 ||, has an added feature, grouping, that we shall discuss later.)

It is interesting to note that in writing a mark number we actually count up to it. For example, in order to write the mark number XXXXX, we must first write X, then writing another X we have XX, and then XXX, and so on.

2.2 Addition of Mark Numbers

There are at least two ways to add mark numbers. One way, just putting the two numbers together with the WORD operation, is exceptionally simple.

```
←TO ADD /M/ AND /N/
>1Ø OUTPUT WORD OF /M/ AND /N/
>END
←
```

With this algorithm, the two numbers are brought together into one. For example, with the bag of pebbles representation, adding two bags of pebbles is just pouring the pebbles into one bag. With mark numbers, adding is just putting all the X's from two words into one word. The first demonstration program, DEMO-ADDA, (see appendix for all demonstration programs) illustrates the two words coming together into one.

If we use the "notches on the gun" representation, the above method doesn't work very well. There is no way to take the notches on two guns and somehow move them on to a single gun. What we can do however is take a new gun and make notches on it, at the same time checking off the notches on one of the old guns. When all of these are checked, we continue with the other old gun. At the end, the new gun will have the sum of the notches from the two old guns. The idea here is that to add /N/ and /M/ we first count to /N/ and then continue counting /M/ more steps. The second demonstration program, DEMO-ADDB, illustrates this method of adding.

This method can be made more efficient by eliminating the first counting up to /N/. Instead, we can just start there and count /M/ more. For example, if we were given two sheets of paper,

one with the mark number for 1000 and the other with the mark number 37 and were told to hand in the sum, we could take a clean sheet and count to 1000 and then 37 more. On the other hand, we could take the sheet that already has 1000 written on it and just count 37 more.

2.3 Comparing Mark Numbers

It's reasonable to think, as we mentioned before, that numbers were invented for some purpose like making sure that the number of sheep that returned from pasture was the same as the number that set out. In other words, two numbers, the number that left and the number that returned, had to be compared to see if they were the same. If we want to compare two mark numbers on the computer, we can use the built-in operation IS. IS /M/ /N/ will output "TRUE" if the numbers are the same and "FALSE" if they are different. It has one drawback compared to any scheme the shepherd might have used to compare his two numbers. It doesn't tell which of the two is bigger! In fact, it's difficult to think of a method the shepherd could have used to find out whether two numbers were the same that wouldn't also tell which one is actually bigger.

Let's look at some algorithms for comparing mark numbers and their programs.

(1) Comparing the numbers /M/ and /N/ will give the same answer as comparing one less than /M/ with one less than /N/. But if we keep reducing the problem this way, eventually either /M/ or /N/ will become /EMPTY/. Then we'll know the other is bigger.

```
←TO COMPARE /M/ AND /N/
>1Ø TEST EMPTYP OF /M/              (Is /M/ empty?)
>2Ø IF TRUE OUTPUT "SECOND" (If so, the second number /N/ is bigger)
>3Ø TEST EMPTYP OF /N/              (Similarly, is /N/ empty?)
>4Ø IF TRUE OUTPUT "FIRST"  (If so, the first number /M/ is bigger)
>5Ø OUTPUT COMPARE OF
     (BUTFIRST OF /M/) AND
     (BUTFIRST OF /N/)       (Get the answer for the reduced
>END                          problem.)
←
```

(2) We can start counting at /EMPTY/ (the zero mark number) and keep on until we get to one of the two numbers that we are comparing. The first one we reach will be the smaller number.

```
←TO COMPARE /M/ AND /N/
>1Ø MAKE                          (Start counting from /EMPTY/)
      NAME:  "COUNT"
      THING:  /EMPTY/
>2Ø TEST IS /COUNT/ /M/
>3Ø IF TRUE OUTPUT "SECOND"  (If /COUNT/ reaches /M/, /N/ is
                               bigger)
>4Ø TEST IS /COUNT/ /N/      (If /COUNT/ reaches /N/, /M/ is
>5Ø IF TRUE OUTPUT "FIRST"     bigger)
>6Ø MAKE                     (Otherwise, /COUNT/ is not big
      NAME:  "COUNT"          enough yet, so add one to it)
      THING:  WORD OF /COUNT/ AND "X"
>7Ø GO TO LINE 2Ø                        (Try again)
>END
←
```

(3) We can count backwards from the first number and see if we get to the second. If we do, the first number must have been the larger one.

```
←TO COMPARE /M/ AND /N/
>1Ø MAKE                          (Start backing up on /M/)
      NAME"  "M"
      THING:  BUTFIRST OF /M/
>2Ø TEST IS /M/ /N/
>3Ø IF TRUE OUTPUT "FIRST"        (We've come down to /N/ so /M/ must
                                   have started out larger)
>4Ø TEST EMPTYP OF /M/
>5Ø IF TRUE OUTPUT "SECOND"       (We've counted all the way down to
                                   /EMPTY/ without passing /N/, so
                                   /N/ must have been larger)
>6Ø GO TO LINE 1Ø                 (Try again)
>END
←
```

(4) We can type out the two numbers by typing first an X from the first number, then an X from the second number, and repeating this until one of the numbers is completely typed. The one that is finished first is the smaller number. The procedure DEMO-COMPARE uses this scheme.

```
←DEMO-COMPARE "XXX" AND "XXXXX"
FIRST     SECOND
  X         X
  X         X
  X         X
            X          (Now the program knows which number is
            X           larger but it finishes typing the larger
                        one anyway)
SECOND                 (... and announces the answer)
←
```

These are just some of the ways mark numbers can be compared. Possibly other ways will occur to students. One interesting way that works for people, but not for the computer, is to simply *look at* the two numbers together. The problem is that the operation of "looking at" is complex, for people as well as computers. But people have a working program for this, and the computer does not.

Even people have problems looking at some numbers. In fact, if we consider very large numbers, a person can't compare them at a glance and he would be forced to resort to some method similar to the programs we've just written. This is exactly what the prehistoric shepherd had to do to compare the number represented by the sheep with the number represented by the pebbles.

One small bug in all of the COMPARE procedures above is their behavior when the two inputs are the same. They will still output either "FIRST" or "SECOND". This can be easily fixed by adding a special check for equality and outputting, say, "EQUAL" if it is true. The first procedure might then look like this.

```
←TO COMPARE /M/ AND /N/
>5 TEST IS /M/ /N/
>6 IF TRUE OUTPUT "SAME"
>1Ø TEST EMPTYP /M/              (The rest is the same as before)
>2Ø IF TRUE OUTPUT "SECOND"
>3Ø TEST EMPTYP /N/
>4Ø IF TRUE OUTPUT "FIRST"
>5Ø OUTPUT COMPARE OF BUTFIRST OF /M/
    AND BUTFIRST OF /N/
>END
←
```

2.4 Subtraction of Mark Numbers

The difference between two numbers can be thought of in several ways. With mark numbers the most obvious way is to think of the difference between two numbers as the amount one number is bigger than the other. If we have two mark numbers,

XXXXXXXX and
XXXXX

then we can see that the difference is the number circled

XXXXXXXX
XXXXX

That is XXX. We can alter the first compare program so that

instead of outputting "FIRST" or "SECOND" it outputs the difference between the numbers. We'll use the fact that the difference between two numbers is the same as the difference between the two numbers both reduced by one. That is, the difference between XXX and XXXXX is the same as the difference between XX and XXXX which is the same as the difference between X and XXX which is the same as the difference between /EMPTY/ and XX. But, when one of the numbers is /EMPTY/, the difference is just the other number. So a program can be written

```
←TO DIFFERENCE /M/ AND /N/
>1Ø TEST EMPTYP OF /M/
>2Ø IF TRUE OUTPUT /N/          (If /M/ is /EMPTY/, the difference
                                 between /M/ and /N/ is all of /N/)
>3Ø TEST EMPTYP OF /N/
>4Ø IF TRUE OUTPUT /M/          (Similarly in the opposite case)
>5Ø OUTPUT DIFFERENCE OF        (Neither is /EMPTY/ so the answer
    (BUTFIRST OF /M/) AND        is the same as the difference of
    (BUTFIRST OF /N/)            one less than each number)
>END
←
```

Notice that this program is essentially the same as the very first COMPARE program, except that different things are output in lines 20 and 40. The COMPARE program actually finds how big the difference between the numbers is, but it does not output that; it merely outputs either "FIRST" or "SECOND" to indicate which of the numbers is bigger.

An important characteristic of the difference between two numbers is that if we add this to the smaller number we get the larger one. We can use this fact to write a different kind of difference program. We'll call this one SUBTRACT. SUBTRACT has two inputs. It starts from /EMPTY/ and counts until it finds a number which, when added to the smaller input, gives the larger one.

```
←TO SUBTRACT /LARGER/ AND /SMALLER/
>1Ø MAKE
      NAME:  "COUNTER"
      THING:  /EMPTY/                    (Start the count at /EMPTY/)
>2Ø TEST IS /LARGER/ (ADD OF /COUNTER/
   AND /SMALLER/)
>3Ø IF TRUE OUTPUT /COUNTER/             (/COUNTER/ is the difference
                                          if /LARGER/ is the sum of
                                          /COUNTER/ and /SMALLER/)
>4Ø MAKE
      NAME:  "COUNTER"
      THING:  WORD OF /COUNTER/ AND "X"  (/COUNTER/ wasn't the
                                          difference so now
                                          increase it by one ...)
>5Ø GO TO LINE 2Ø                         (... and try again)
>END
←
```

This program is a little different from DIFFERENCE. With DIFFERENCE we could write the inputs in either order.

```
←PRINT DIFFERENCE OF "XXXX" AND "XX"
XX
←PRINT DIFFERENCE OF "XX" AND "XXXX"
XX
←
```

But, with SUBTRACT the larger number must come first. If we were to ask the computer to PRINT SUBTRACT OF "XX" AND "XXXX", it would try to find a number which it could add to "XXXX" to get "XX". It will check /EMPTY/, then X, then XX, then XXX, and so on. It will never find a number that works and so it will continue searching until someone interrupts it.

We could have avoided this "bug" by adding the lines

```
2 TEST IS COMPARE OF /LARGER/ AND /SMALLER/ "SECOND"
3 IF TRUE OUTPUT SUBTRACT OF /SMALLER/ AND /LARGER/
```

These lines cause the two inputs to exchange if /SMALLER/ is larger than /LARGER/.

Discussion about this "deficiency" of the natural numbers (or counting numbers or positive integers), that there is no number which can be added to "XXXX" so as to give "XX", will be used later in motivating the construction of negative numbers.

3. Compact Numbers

3.1 Grouping Mark Numbers

When mark numbers are used in scorekeeping these days, the most common form is 𝍸 𝍸 ||| where the marks are grouped into bunches of five marks each. A possible origin of this technique is finger counting. If we count on our fingers up to a large number, we'd be apt to think of the number as so many hands and so many fingers left over. Suppose a prehistoric general wanted to count the soldiers in his army. He could get an aide to help him. The aide would stand by and raise one finger as each soldier walked by him. After ten soldiers passed, all of the aide's fingers would be raised and the general would call another aide. This aide would count the next ten men on his fingers, and then the general would call still another aide. When the counting is over, the number of soldiers is represented by the aides used to determine it -- Marcus, Attila, Xenophon, Jura, Albert, and part of Hector (his left hand and thumb). This is much harder to remember than "fifty six" but it is much easier to remember than "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX".

We can write a procedure GROUP-COUNT that will count using mark numbers, but leave a space after every ten marks. Then, each group of marks stands for one man's fingers. One way to write this program is to use a subprocedure that types out ten marks. Then the main procedure will only have to call this subprocedure,

then type a space, and then go back to the beginning and start over. If this were all we wanted to do, then we could just use TYPE "XXXXXXXXXX" as the subprocedure and the program would work. (Remember TYPE is like PRINT except that it doesn't carriage return.)

```
←TO GROUP-COUNT
>1Ø TYPE "XXXXXXXXXX"
>2Ø TYPE /BLANK/                (Types a blank space)
>3Ø GO TO LINE 1Ø
>END
←

←GROUP-COUNT
XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX   (and so on)
```

This would be a more reasonable program if it actually counted something instead of just typing X's at high speed. One idea is to pause before typing a mark and wait until the ENTER key on the teletype is pressed, then typing one mark and waiting again. This way the computer will be counting the number of times the ENTER key is pressed.

The way to make the computer wait until ENTER is pressed is to use the REQUEST operation. For example,

```
←TO COUNT-ONE
>1Ø REQUEST        (The computer waits until the ENTER or RETURN
                    key is pressed)
>2Ø TYPE "X"       (... and then types "X")
>3Ø GO TO LINE 1Ø (... and then goes back to the beginning)
>END
←
```

This program counts in ordinary, ungrouped mark numbers. If we modify it so that it does the two commands REQUEST and TYPE "X" exactly ten times and then stops, we can use it in place of line 1Ø in GROUP-COUNT. It is easy to extend COUNT-ONE in this way.

All we need to do is write

```
←TO COUNT-TEN
>1Ø REQUEST
>2Ø TYPE "X"
>3Ø REQUEST
>4Ø TYPE "X"
>5Ø REQUEST
>6Ø TYPE "X"
>7Ø REQUEST
>8Ø TYPE "X"
>9Ø REQUEST
>1ØØ TYPE "X"
>11Ø REQUEST
>12Ø TYPE "X"
>13Ø REQUEST
>14Ø TYPE "X"
>15Ø REQUEST
>16Ø TYPE "X"
>17Ø REQUEST
>18Ø TYPE "X"
>19Ø REQUEST
>2ØØ TYPE "X"
>END
←
```

Now if we try COUNT-TEN it will count and write ten marks, so we can put it into GROUP-COUNT, as follows.

```
←TO GROUP-COUNT
>1Ø COUNT-TEN
>2Ø PRINT /BLANK/
>3Ø GO TO LINE 1Ø
>END
←
```

COUNT-TEN can be improved from a programming point of view just as SINGTWO was improved to become SINGALOT in the introductory unit. We can write COUNT-TEN so that it takes an input (a mark number) and types the number of marks of the input. If the input is /EMPTY/, it will stop. Otherwise, it will type one X and then shorten the input by one. Then it will start again with this shorter input.

```
←TO COUNTALOT /INPUT/
>1Ø TEST EMPTYP OF /INPUT/    (Are there any marks left in /INPUT/?)
>2Ø IF TRUE STOP              (If not, STOP)
>3Ø REQUEST
>4Ø TYPE "X"                  (Otherwise, COUNT-ONE)
>5Ø MAKE                      (Shorten the input by one)
      NAME:  "INPUT"
      THING:  BUTFIRST OF /INPUT/
>6Ø GO TO LINE 1Ø             (and go back to the beginning)
>END
←
```

We can shorten COUNTALOT a little more by using recursion. To do this, we notice that after we've typed the first X the work we have left to do is exactly what COUNTALOT (BUTFIRST OF /INPUT/) does. That is, we want to type one fewer X's than before. So we can write,

```
←TO COUNTALOT /INPUT/
>1Ø TEST EMPTYP OF /INPUT/
>2Ø IF TRUE STOP
>3Ø REQUEST
>4Ø TYPE "X"
>5Ø COUNTALOT (BUTFIRST OF /INPUT/)  (We've typed one X.  Now
                                       we'll type the rest of them)
>END
←
```

Now that we've changed COUNTALOT we have to go back to GROUP-COUNT. COUNTALOT needs an input. That input is the number of marks in each group, which could be any number. We decided to use ten, to represent the number of fingers on a man. So we write,

```
←TO GROUP-COUNT
>1Ø COUNTALOT "XXXXXXXXXX"
>2Ø TYPE /BLANK/
>3Ø GO TO LINE 1Ø
>END
←
```

We can, if we like, use groups of some size other than ten. We would only need to change line 1Ø of GROUP-COUNT. We will use

ten mark groups because they lead nicely into base ten number systems. If we were actually adopting group notation, however, it would probably be handier to use groups of five. The reason for this smaller grouping is that a reader can see at a glance whether a group has one, two, three, four, or five marks in it, while it is not so easy to distinguish groups of eight, nine, and ten marks. For groups that large it is helpful, even for people, to use some kind of discrimination procedure.

3.2 Conversion of Mark Numbers to Grouped Numbers

Since it's easier to read a mark number when it is grouped, we'll consider the problem of writing a procedure that takes an ordinary mark number and converts it to a grouped number. For example

```
←PRINT GROUP OF "XXXXXXXXXXXXX"
XXXXXXXXXX XXX
←
```

One of the easiest ways we've found to write this procedure is by using two subprocedures, FIRSTTEN and BUTFIRSTTEN. As the names suggest, these procedures are like FIRST and BUTFIRST except that they deal in groups of ten letters at a time. FIRSTTEN of a word is the first ten letters of the word; BUTFIRST-TEN of a word is the rest of the word, that is, the whole word except for the first ten letters.

We'll write GROUP first, assuming that we already have FIRSTTEN and BUTFIRSTTEN, and then we'll write those two procedures. One advantage to doing the writing in this order is that we can check out the soundness of the superstructure before investing time in working out its parts in detail. Further, we may discover, while writing GROUP, some special things FIRSTTEN or BUTFIRSTTEN should

do to make GROUP easier, whereas writing the two subprocedures first is not likely to help us in writing GROUP.

The basic idea behind the procedure GROUP is recursive. We'll reduce the problem of grouping a large word to that of grouping a smaller word. We'll keep this up until the problem is reduced to grouping the /EMPTY/ word. (That's easy. The result is /EMPTY/.)

In order to reduce grouping a large word to grouping a smaller one, we'll take the first ten marks of the word and put them into one group and then group the BUTFIRSTTEN of the word. To attach these two parts together as a group number, we can use the operation SENTENCE. SENTENCE OF (FIRSTTEN OF /WORD/) AND (GROUP OF BUTFIRSTTEN OF /WORD/) should take the first ten marks of /WORD/ and put a space between them and GROUP of the rest of the word. So GROUP is

```
←TO GROUP /WORD/
>1Ø TEST EMPTYP OF /WORD/
>2Ø IF TRUE OUTPUT /EMPTY/      (If we've reduced /WORD/ all the
                                 way to /EMPTY/, then output the
                                 answer /EMPTY/)
>3Ø OUTPUT SENTENCE OF          (Otherwise, to get the answer, take
    (FIRSTTEN OF /WORD/) AND     the first ten, and GROUP of all
    (GROUP OF BUTFIRSTTEN OF     the rest and make a sentence of
    /WORD/)                      them.)
>END
←
```

Now we can write BUTFIRSTTEN. All we need to do is peel off the first ten marks. BUTFIRST OF /N/ will peel off one mark (BUTFIRST OF "XXX" is "XX"). BUTFIRST OF BUTFIRST OF /N/ will peel off two marks since what it says is take the BUTFIRST of whatever BUTFIRST of /N/ is. And finally, if we use ten BUTFIRST's, we can peel away ten marks.

```
←TO BUTFIRSTTEN /N/
>1Ø OUTPUT BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF /N/
>END
←
```

In writing FIRSTTEN we can use an important fact. If a word has ten or more marks in it, FIRSTTEN of the word is "XXXXXXXXXX". We don't have to actually use the first ten letters of the word itself by peeling off the last ones, or by using WORD to put together the first letter and the second and the third and so forth. We *would* have to do this if we wanted FIRSTTEN to work on *arbitrary* words (like "ABCDEFGHIJKLMNOP" or "ANTIDISESTABLISHMENTARIANISM"). But, since we're working with mark numbers, all of the letters in our words are X's.

What should FIRSTTEN do if a word has *fewer than* ten marks? A reasonable action would be to output the whole word. Reasonableness isn't an appropriate criterion, however, if it does not lead to our goal. Will this action by FIRSTTEN cause GROUP to work properly? Before we check that, let's take a look again at BUTFIRSTTEN. When we wrote that procedure, we didn't consider whether inputs had fewer than ten marks. The way we've written it, BUTFIRSTTEN will output /EMPTY/ for an input of ten or fewer marks. (Because BUTFIRST of /EMPTY/ is /EMPTY/.) So in the case of BUTFIRSTTEN we made a decision about small mark numbers more or less accidentally. We may have to go back and change it if it doesn't work out properly in GROUP.

Let's see what happens when GROUP gets an input smaller than ten X's, say "XXXX". First, "XXXX" isn't /EMPTY/ so we'll output the sentence of FIRSTTEN OF "XXXX" and GROUP OF BUTFIRSTTEN OF "XXXX". Assuming we've tentatively made the decisions for FIRSTTEN and

BUTFIRSTTEN that we felt were reasonable, then FIRSTTEN OF "XXXX" is "XXXX" and BUTFIRSTTEN OF "XXXX" is /EMPTY/. So we're outputting sentence of "XXXX" and GROUP OF /EMPTY/. But GROUP OF /EMPTY/ is /EMPTY/ because of lines 1Ø and 2Ø in GROUP. So we're outputting sentence of "XXXX" and /EMPTY/. That is just "XXXX". This says that when the input is a word smaller than ten marks, if we write FIRSTTEN to output the whole word and BUTFIRSTTEN to output /EMPTY/, then GROUP of the input word will be the word itself. Thus, GROUP OF "XXXX" will be "XXXX". But this is exactly what GROUP is supposed to do in this case, and what happens here with "XXXX" is typical of what should happen to any mark number with ten or fewer marks.

Thus, we'll write FIRSTTEN so that it outputs the whole word if the input has fewer than ten marks. The procedure will test if there are at least ten marks. If there are, it will output "XXXXXXXXXX"; if there aren't, it will output the whole word. A way to test whether there are at least ten marks is to peel off nine, using nine BUTFIRST's and then see if there are any left.

```
←TO FIRSTTEN /INPUT/
>1Ø TEST EMPTYP OF BUTFIRST OF    (Is there anything left after
    BUTFIRST OF BUTFIRST OF        peeling off nine marks?)
    BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF
    /INPUT/
>2Ø IF FALSE OUTPUT "XXXXXXXXXX" (If so, /INPUT/ must have had at
                                   least ten marks since it wasn't
                                   /EMPTY/ after nine were peeled
                                   away)
>3Ø IF TRUE OUTPUT /INPUT/       (If not, /INPUT/ has fewer than
                                   ten marks so output the whole
                                   input word)
>END
←
```

Now GROUP should work. Let's test it.

```
←PRINT GROUP OF "XXXXXXXXXXXXXXXXXXXXXXX"
XXXXXXXXXX XXXXXXXXXX XXX
←
```

It worked with that input. That's a good sign but it's not completely convincing in showing that GROUP will work for all inputs. How can we make a test that will be more convincing? Of course, we can't simply test *every* possible input since there are an infinity of them. What we can do is to test a representation of each different type of input. That is, each input which the procedure seems to handle differently. In this case we can properly use:

"XXXXXX" An input with fewer than ten marks.
"XXXXXXXXXX" Exactly ten marks.
"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" An input that groups evenly with no marks left over.
"XXXXXXXXXXXXXXXXXXXXXXX" Several groups and some left over.

If GROUP works with these four inputs, we can be fairly sure it will always work; not perfectly sure, since we might have overlooked an important type, but nearly so. This way of choosing test inputs is much more effective in finding bugs than choosing test inputs at random.

```
←PRINT GROUP OF "XXXXXX"
XXXXXX
←PRINT GROUP OF "XXXXXXXXXX"
XXXXXXXXXX
←PRINT GROUP OF "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX
←PRINT GROUP OF "XXXXXXXXXXXXXXXXXXXXXXX"
XXXXXXXXXX XXXXXXXXXX XXX
←
```

So GROUP probably works. There is an additional feature in LOGO that will let us watch more closely and exactly now GROUP works. This feature, called TRACE, will cause the teletype to print the inputs of GROUP each time it is called. For example,

```
←TRACE GROUP                          (This turns on the trace feature)
←PRINT GROUP OF "XXXXXXXXXXXXXX"
GROUP OF "XXXXXXXXXXXXXX"             (GROUP is first called with this
                                       input)
  GROUP OF "XXXX"                     (Next, GROUP is called with BUTFIRST-
                                       TEN of that input.  This calling
                                       happens at line 3Ø in GROUP)
    GROUP OF ""                       (To compute GROUP OF "XXXX" the com-
                                       puter needs to output sentence of
                                       FIRSTTEN of "XXXX" and GROUP of
                                       BUTFIRSTTEN of "XXXX", that is GROUP
                                       of /EMPTY/, or as written, GROUP OF
                                       "")
    GROUP OUTPUTS ""                  (GROUP OF "" can be computed immedi-
                                       ately, line 2Ø, and doesn't call
                                       GROUP again)
  GROUP OUTPUTS "XXXX"                (Now that GROUP OF "" has been com-
                                       puted, GROUP OF "XXXX" can output)
GROUP OUTPUTS "XXXXXXXXXX XXXX"       (Finally, GROUP OF "XXXXXXXXXXXXXX"
                                       can output since GROUP OF "XXXX" is
                                       now known)
XXXXXXXXXX XXXX                       (PRINT types out the final result)
←ERASE TRACE GROUP                    (This turns off the trace feature)
←
```

This TRACE feature is actually most useful when a procedure doesn't work properly at first and we want to know what is wrong. We turn on the TRACE feature and run our procedure with an input that doesn't work. Then, by looking at the printout we can often see where things are going wrong.

Notice the way inputs and outputs pair in the TRACE printout.

```
←TRACE GROUP
←PRINT GROUP OF "XXXXXXXXXXXXXX"
┌GROUP OF "XXXXXXXXXXXXXX"
| ┌GROUP OF "XXXX"
| | ┌GROUP OF ""
| | └GROUP OUTPUTS ""
| └GROUP OUTPUTS "XXXX"
└GROUP OUTPUTS "XXXXXXXXXX XXXX"
 XXXXXXXXXX XXXX
←
```

If the outermost GROUP gave the wrong answer, we could check the others to see where the error started. In this way we can usually find a very simple case where the procedure goes wrong. Then, by pretending to be the computer, we can examine that case in great detail and find the bug.

3.3 A Compact Representation of Grouped Numbers

So now we can have mark numbers typed out in groups of ten. We can go a step further and instead of typing out ten X's in each group, we can use another letter to represent a whole group. For example, if we choose T to stand for ten marks, then the number XXXXXXXXXXXXXXXXXXXXXXX, or XXXXXXXXXX XXXXXXXXXX XXX, would be written TTXXX; that is, two groups of ten marks and XXX left over. One procedure for doing this is very similar to the one used in writing GROUP. In fact, we will be able to use the same programs we used there, with some minor changes. First, wherever GROUP had a word of ten X's we now want a T. The words of ten X's always come from line 2Ø of FIRSTTEN. So we'll change that to output T instead of XXXXXXXXXX.

```
←TO FIRSTTEN /INPUT/
>1Ø TEST EMPTYP OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF /INPUT/
>2Ø IF FALSE OUTPUT "T"      (Instead of the ten X's)
>3Ø IF TRUE OUTPUT /INPUT/   (/INPUT/ has fewer than ten marks so
                              the X's are output, as before,
                              instead of a T)
>END
←
```

Now we'll try GROUP with this changed subprocedure.

```
←PRINT GROUP "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
T T T XXXXXX
←
```

This is almost exactly what we want. The only things wrong with it are the spaces. These are due to the line in GROUP that says OUTPUT SENTENCE OF ... . If we change SENTENCE to WORD, the various parts will be put together without spaces. While we make this change let's also change the name of the procedure from GROUP to something else, since its function has changed. It now converts a mark number to a different kind of number than a grouped mark number. We'll use the name COMPACT for the procedure and call numbers made up of these T's and X's *compact numbers*.

```
←TO COMPACT /WORD/
>1Ø TEST EMPTYP OF /WORD/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF FIRSTTEN OF /WORD/ AND COMPACT OF
    BUTFIRSTTEN OF /WORD/
>END
←
```

COMPACT is exactly the same as GROUP except that WORD replaces sentence in line 3Ø. Let's try it out.

```
←PRINT COMPACT OF "XXX"
XXX
←PRINT COMPACT OF "XXXXXXXXXX"
T
←PRINT COMPACT OF "XXXXXXXXXXXXXXXXXXXXX"
TTXX
←PRINT COMPACT OF "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
TTT
←
```

So COMPACT seems to work.

Compact numbers have one big advantage over mark numbers. They are much smaller, that is, they have fewer marks. The mark number that corresponds to TTTTTTTTXXX wouldn't fit on one line. Because of their size, compact numbers are easier to write and easier to read. On the other hand, counting, adding, subtracting, and comparing are somewhat harder with compact numbers. TX is a bigger number than XXXXXXXXX despite appearances. Also, we can no longer count just by adding a mark to the preceding number. Sometimes it works, but every tenth time we have to rewrite the number, changing the nine X's into one T.

The advantages above seem, in practice, to far outweigh the disadvantages. As far back as history goes men used some form of compact numbers, usually with more than two different marks (Roman numerals or Babylonian cuneiform for example). Roman numerals, in fact, have been going out of use only gradually over the last thousand years and are still used today when a second number system is needed. Examples are for numbering chapters in a book, pages in an introduction, or topics in an outline.

3.4 Conversion of Compact Numbers to Mark Numbers

We have a program, COMPACT, that converts mark numbers to compact numbers. We could also use a program that goes in the opposite

direction, converting compact numbers to mark numbers. This is useful because mark numbers are, in many ways, easier to deal with than compact numbers. They are easier to compare and to add, for example. (We'll see later that mark numbers are also easy to multiply and divide.)

A simple idea for writing this program is to go through the compact number, leaving the X's alone and changing each T to XXXXXXXXXX. What we can do is take a T away from the front of the compact number and add XXXXXXXXXX to the back. Then we can do this over and over again until there are no more T's at the front of the word. For example, TTX would be changed to TXXXXXXXXXXX by taking off the first T and adding on the ten X's. Then TXXXXXXXXXXX would become XXXXXXXXXXXXXXXXXXXXX. This number doesn't begin with T, so we're done.

```
←TO UNCOMPACT /N/
>1Ø TEST IS FIRST OF /N/ "T"     (Does the number begin with T)
>2Ø IF FALSE OUTPUT /N/          (It doesn't, so all the T's must
                                  be gone.  So it is a mark number
                                  and we should output it)
>3Ø MAKE                         (Remove the first T)
      NAME:  "N"
      THING:  BUTFIRST OF /N/
>4Ø MAKE                         (And put ten X's on the end)
      NAME:  "N"
      THING:  WORD OF /N/ AND    (Repeat the process with the
              "XXXXXXXXXX"        partially converted number, taking
>5Ø GO TO LINE 1Ø                 care of the next T if there is one)
>END
←
```

Now we should test UNCOMPACT. XXXXX and TTXX and TTT seem to be representative cases.

```
←PRINT UNCOMPACT OF "XXXXX"
XXXXX
←PRINT UNCOMPACT OF "TTXX"
XXXXXXXXXXXXXXXXXXXX
←PRINT UNCOMPACT OF "TTT"
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
←
```

An amusing way to test UNCOMPACT is to notice that COMPACT OF UNCOMPACT OF /X/ should be /X/ and UNCOMPACT OF COMPACT OF /X/ should also be /X/. (Provided, of course, that /X/ is a compact number in the first case and a mark number in the second.)

```
←PRINT COMPACT OF UNCOMPACT OF "TTTXX"
TTTXX
```

Since UNCOMPACT OF "TTTXX" is "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" and COMPACT of that is "TTTXX".

```
←PRINT UNCOMPACT OF COMPACT OF "XXXXXXXXXXXXX"
XXXXXXXXXXXXX
```

Since COMPACT OF "XXXXXXXXXXXXX" is "TXXX" and UNCOMPACT OF "TXXX" is "XXXXXXXXXXXXX".

It is also interesting to try COMPACT OF UNCOMPACT and UNCOMPACT OF COMPACT on words that aren't numbers (for example, "ABCDEFGHIJKLMNOPQRSTUVWXYZ") to see what they do. A good problem is to describe exactly which words these two operations (COMPACT OF UNCOMPACT and UNCOMPACT OF COMPACT) leave unchanged. The effect of these operations might be surprising. For example,

```
←PRINT COMPACT OF UNCOMPACT OF "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
TTUVWXYZ
```

The explanation is straightforward. Since "ABCDEFGHIJKLMNOPQRST-UVWXYZ" doesn't begin with "T", UNCOMPACT leaves it unchanged. COMPACT substitutes a "T" for the first ten marks, "ABCDEFGHIJ", and another for the next ten, "KLMNOPQRST", and leaves the last six unchanged.

3.5 Disordered Compact Numbers

So far all the compact numbers we've looked at have had all T's to the left of the X's. But, what about a number like XXXTTX? If we follow the rule that a T stands for ten X's, then that number is the same as XXXXXXXXXXXXXXXXXXXXXX. That is, it is the same as TTXXXX or TXXXXT or several other forms, such as XXXXXXXTXXXXXXX. There are many ways of writing this number, and the order of marks really doesn't matter at all. It doesn't even matter if the number isn't completely compacted. It is always clear which number is meant. (We're ignoring the commonly used subtractive notation of Roman numerals which makes IV the same as IIII. The only important modern use of this is in telling time anyway.)

We shall write an uncompacting procedure to work for any form of compact number. Notice that our old UNCOMPACT doesn't work with most compact numbers. It will convert the T's on the left end of the word but not any others.

```
←PRINT UNCOMPACT OF "XXTTT"
XXTTT
←PRINT UNCOMPACT OF "TXTXT"
XXXXXXXXXXTXT
```

The simplest way to write this procedure is to use recursion. We note that if a word begins with X, then UNCOMPACT OF the word is the same as UNCOMPACT OF BUTFIRST OF that word, with an X stuck on the end. If the word begins with T, then UNCOMPACT of the word is the same as UNCOMPACT OF BUTFIRST OF the word, with XXXXXXXXXX stuck on the end. So, UNCOMPACT works on the word by working on the BUTFIRST of the word. And UNCOMPACT works on the BUTFIRST of the word by working on the BUTFIRST OF BUTFIRST of the word. And so on. With each time around, the word gets smaller.

Eventually it will be /EMPTY/. And UNCOMPACT OF /EMPTY/ is /EMPTY/, so the process terminates.

```
←TO UNCOMPACT /N/
>1Ø TEST EMPTYP OF /N/
>2Ø IF TRUE OUTPUT /EMPTY/     (If /N/ is /EMPTY/, the answer is
>3Ø TEST IS FIRST OF /N/ "X"                          /EMPTY/)
>4Ø IF TRUE OUTPUT WORD OF (UNCOMPACT
    OF BUTFIRST OF /N/) AND "X"     (FIRST is "X", so tack "X"
                                     onto UNCOMPACT of BUTFIRST)
>5Ø OUTPUT WORD OF (UNCOMPACT OF
    BUTFIRST OF /N/) AND
    "XXXXXXXXXX"                  (FIRST is not "X" so it must be T.
                                   Hence tack "XXXXXXXXXX" onto
                                   UNCOMPACT OF BUTFIRST)
>END
←
```

Let's look at this procedure. Obviously it works for empty inputs. If /N/ is /EMPTY/, then UNCOMPACT will output /EMPTY/, the correct response. What happens if /N/ is only one letter long? Well, if /N/ is "X", then line 4Ø is used and UNCOMPACT outputs WORD OF UNCOMPACT OF /EMPTY/ AND "X". But UNCOMPACT OF /EMPTY/ is /EMPTY/ as we just saw, so this is WORD OF /EMPTY/ AND "X" or "X". Thus, UNCOMPACT OF "X" is "X". If /N/ is "T", then line 5Ø is used and UNCOMPACT outputs WORD OF UNCOMPACT OF /EMPTY/ AND "XXXXXXXXXX". That is WORD OF /EMPTY/ AND "XXXXXXXXXX", or "XXXXXXXXXX". So UNCOMPACT OF "T" is "XXXXXXXXXX". UNCOMPACT works for both of the one-letter inputs. We could continue like this, showing that UNCOMPACT works for all two-letter inputs and for all three-letter inputs, and so on. Psychologically this can be very convincing. After a short time (say, when we've proved it for up to seven-letter inputs), anyone but a mathematician probably would be completely assured that UNCOMPACT will always work. The mathematician would ask how we could be sure that the first input for which UNCOMPACT fails isn't some huge word consisting of millions of letters.

Fortunately, there are ways to avoid this problem. One is called mathematical induction (or, in the form we will use, the method of infinite descent). We will show that if there is any compact number that UNCOMPACT doesn't work on, then there is a smaller compact number (BUTFIRST of the original number) on which UNCOMPACT also doesn't work. But, then we'll be able to apply this argument again to the new number and get a still smaller number on which UNCOMPACT won't work. We can continue this way until eventually we'll be down to a one-letter number which UNCOMPACT can't handle. But there are only two one-letter numbers, X and T, and UNCOMPACT works perfectly on both of them. So this method will show that there can't be any compact number that UNCOMPACT won't correctly convert into a mark number.

Well, all we need to show is that if UNCOMPACT doesn't work for some compact number, then it also doesn't work for the BUTFIRST of that number. The number begins with either X or T. If it begins with X, then BUTFIRST of it is a compact number one less than the original number. So UNCOMPACT OF BUTFIRST OF the original number should have one mark less than UNCOMPACT OF the original number. But, the way the procedure for uncompacting works in this case (line 4Ø of UNCOMPACT) is to tack one more X onto the UNCOMPACT OF BUTFIRST. So if UNCOMPACT of the number itself is wrong, then UNCOMPACT of the BUTFIRST of the number must have been wrong.

The same argument is true if the original number began with T. We'd refer to line 5Ø of the procedure and to tacking on XXXXXXXXXX instead of line 4Ø and tacking on X.

This argument will convince the mathematician that UNCOMPACT will work for any compact number. It is a curious fact that the earlier, incomplete argument is much more likely to be convincing

to nonmathematicians. This is probably because few nonmathematicians have any fluency with complex *logical* arguments. Arguments in life situations nearly always hinge on questions of evidence or values rather than subtle logical points.

The reason for choosing mathematical induction for the test of UNCOMPACT is the close relationship between induction and recursion. When we write a recursive procedure, we decide to solve the problem of a large input by solving the same problem for a smaller input. Eventually the input becomes small enough so that the solution is trivial (usually we let the input get down to /EMPTY/). With a related form of mathematical induction, we say that if the solution is wrong for a large input it must be wrong for a smaller input. Then it must be wrong for a still smaller input. Finally, the solution must be wrong for the case of /EMPTY/. But, we can check that case and see that is correct. So, tne solution can't be wrong for any input.

Tracing will show how the UNCOMPACT procedure makes the problem simpler and simpler until it can finally solve it.

```
←TRACE UNCOMPACT
←PRINT UNCOMPACT OF "XTX"
UNCOMPACT OF "XTX"                  (The original problem)
  UNCOMPACT OF "TX"                 (We can solve the original problem
                                     if we know the answer to this one.
                                     The solution to the original prob-
                                     lem is WORD OF (UNCOMPACT OF "TX")
                                     AND "X")
    UNCOMPACT OF "X"                (Now we can solve the problem just
                                     above if we can solve this problem.
                                     The solution to the problem above,
                                     UNCOMPACT OF "TX" is WORD OF
                                     (UNCOMPACT OF "X") AND "XXXXXXXXXX",
                                     by line 5Ø of UNCOMPACT.  If we
                                     substitute this into the solution
                                     of the original problem above we get
                                     that, the solution is WORD OF (WORD
                                     OF UNCOMPACT OF "X" AND "XXXXXXXXXX")
                                     AND "X")
```

```
      UNCOMPACT OF ""              (BUTFIRST OF "X" is /EMPTY/, also
                                    written "".  By line 40 of UNCOMPACT
                                    we have UNCOMPACT OF "X" is WORD OF
                                    (UNCOMPACT OF "") AND "X")
      UNCOMPACT OUTPUTS ""         (The problem UNCOMPACT OF "" is al-
                                    ready as simple as possible.  The
                                    answer is "")
    UNCOMPACT OUTPUTS "X"          (Now that UNCOMPACT OF "" is solved
                                    UNCOMPACT OF "X" is known.  It was
                                    just WORD OF (UNCOMPACT OF "") AND
                                    "X")
  UNCOMPACT OUTPUTS "XXXXXXXXXXX"  (Remember that UNCOMPACT OF "TX"
                                    was WORD OF (UNCOMPACT OF "X") AND
                                    "XXXXXXXXXX".  Now that UNCOMPACT
                                    OF "X" is solved, UNCOMPACT OF "TX"
                                    is known)
UNCOMPACT OUTPUTS "XXXXXXXXXXXX"   (Finally the original problem
                                    is solved now that UNCOMPACT OF
                                    "TX" is known)
XXXXXXXXXXXX                       (And the result is printed)
←
```

Despite all these discussions about UNCOMPACT, we should still test it with some representative inputs just to make triply sure it works.

```
←PRINT UNCOMPACT OF "XT"
XXXXXXXXXX
←PRINT UNCOMPACT OF "XXXXXX"
XXXXXX
←PRINT UNCOMPACT OF "TT"
XXXXXXXXXXXXXXXXXXX
←
```

We can write programs to do arithmetic with compact numbers by using UNCOMPACT to convert the compact numbers into mark numbers, using the old mark number procedures ADD, SUBTRACT, and so on, and then using COMPACT to convert the answer back into a compact number. This method would work but it is inefficient. It preserves the advantages of compact numbers for the people using the computer but not for the computer itself. It would be silly

to worry about being "fair" to the computer except that the more work we make the computer do the longer it takes. And long waits for an answer are annoying to people. So it is to our advantage to allow the computer to operate efficiently.

In handling compact numbers, it is a great convenience to be able to separate the T's and the X's. Suppose we have two procedures, T'S OF /N/ and X'S OF /N/ (the ' can be used in the name of a LOGO procedure just as if it were a letter) that work as follows:

```
←PRINT T'S OF "XXTTXT"
TTT
←PRINT X'S OF "XXTTXT"
XXX
←PRINT T'S OF "T"
T
←PRINT X'S OF "T"
      (/EMPTY/ is printed)
←
```

T'S outputs a word made up of all the T's in its input and X'S does the same for the X's. These two procedures can be used for straightening out disordered compact numbers, for comparing compact numbers (since we have to compare the T's and then, if both have the same number of T's, compare the X's), and so forth.

We'll write T'S and X'S. Let's do T'S first. Then X'S should turn out to be a trivial modification of T'S. We'll write T'S using recursion. All we need to do is to notice that if a word begins with X, then T'S OF that word is the same as T'S OF BUTFIRST OF the word. If the word begins with T, then T'S OF the word is just T'S OF BUTFIRST OF the word, with an extra T stuck on at the end. That is, WORD OF (T'S OF BUTFIRST OF the word) AND "T". So whether the word begins with T or X, we can reduce the problem to getting T'S OF BUTFIRST OF the word. We can keep doing this

until the problem is reduced to finding T'S OF /EMPTY/. This is a trivial question and we can immediately give the answer, /EMPTY/.

```
←TO T'S /N/
>1Ø TEST EMPTYP OF /N/          (Have  we gotten down to the trivial
                                 case yet?)
>2Ø IF TRUE OUTPUT /EMPTY/      (If so, the answer is /EMPTY/; there
                                 are no T's in the empty word)
>3Ø TEST IS FIRST OF /N/ "T"
>4Ø IF TRUE OUTPUT WORD OF (T'S (If /N/ begins with T, then the
    OF BUTFIRST OF /N/) AND "T"  answer is T'S OF BUTFIRST OF /N/
                                 with one more T stuck on)
>5Ø OUTPUT T'S OF BUTFIRST OF   (Otherwise the first letter of /N/
    /N/                          isn't T so the answer is just T'S
                                 OF BUTFIRST OF /N/.  We can just
                                 forget the first letter)
>END
←
```

And, of course, we'll test the procedure.

```
←PRINT T'S OF "XXX"
      (The empty word, since there are no T's in XXX)
←PRINT T'S OF "TT"
TT
←PRINT T'S OF "XTTXTX"
TTT
←
```

The procedure X'S can be written exactly like T'S. We need only replace "T" in lines 3Ø and 4Ø by "X".

```
←TO X'S /N/
>1Ø TEST EMPTYP OF /N/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø TEST IS FIRST OF /N/ "X"
>4Ø IF TRUE OUTPUT WORD OF (X'S OF BUTFIRST OF /N/) AND "X"
>5Ø OUTPUT X'S OF BUTFIRST OF /N/
>END
←
←PRINT X'S OF "TTXTX"
XX
```

3.6 Standard Form

Now that we have the procedures T'S and X'S, it is a simple matter to write some compact number manipulators. For example, while it is true that order has no effect on the value of a compact number, it is also true that it is easier to read compact numbers if they are written in some standard form (say, all T's followed by all X's). So, let's write a procedure to convert any compact number to standard form. All the procedure needs to do is put the T's at the beginning and the X's at the end.

```
←TO STANDARDIZE /NUMBER/
>1Ø OUTPUT WORD OF (T'S OF /NUMBER/)
    AND (X'S OF /NUMBER/)
>END
←
```

We can try STANDARDIZE on some of the disordered compact numbers we mentioned before.

```
←PRINT STANDARDIZE OF "XTXTXX"
TTXXXX
←PRINT STANDARDIZE OF "XXXTT"
TTXXX
←PRINT STANDARDIZE OF "XX"
XX
←PRINT STANDARDIZE OF "TTXX"
TTXX
←PRINT STANDARDIZE OF "XXXXXXTXXXXXX"
TXXXXXXXXXXXX
←
```

STANDARDIZE works, up to a point. It does seem to arrange the T's and X's properly but it doesn't do any compacting. This is perfectly reasonable, since we didn't instruct it to do so when we defined the procedure. We could leave STANDARDIZE as it is and truthfully say that we've written what we set out to write, a program that will arrange the T's and X's of a compact number

so that the T's are to the left of the X's. It is an advantage, however, if any two standard compact numbers that stand for the same mark number are exactly the same. (For example, if we wanted to see if two compact numbers were equal, we could just do IS OF STANDARDIZE OF one AND STANDARDIZE OF the other.) So, let's add to our criteria for standard form that there be no more than nine X's in the number. So, a compact number is in standard form only if (a) all the T's are on the left, and (b) there are no more than nine X's. The reader should convince himself that these rules guarantee that two equal compact numbers will be exactly the same.

In order to fix STANDARDIZE so that it will output numbers in our newly-defined standard form, we need to adjust the phrase (X'S OF /NUMBER/). The problem occurs when (X'S OF /NUMBER/) is a word of more than nine X's. COMPACT is a program that will handle this problem. COMPACT will take the word of X's and output a word with nine or fewer X's, the extra X's compacted into T's. As a special piece of fortune, COMPACT puts those extra T's to the left of the X's, just the right place to connect them properly with the T's output from (T'S OF /NUMBER/). So we can write

```
←TO STANDARDIZE /NUMBER/
>1Ø OUTPUT WORD OF (T'S OF /NUMBER/)
    AND (COMPACT OF X'S OF /NUMBER/)
>END
←
```

If we consider STANDARDIZE OF "XXXXXXTXXXXXX", we can see that the procedure will output WORD OF "T" AND "TXX" which is "TTXX".

```
←PRINT STANDARDIZE OF "XXXXXXTXXXXXX"
TTXX
```

And, of course, we should test the new STANDARDIZE on a representative group of compact numbers.

```
←PRINT STANDARDIZE OF "TT"
TT
←PRINT STANDARDIZE OF "XX"
XX
←PRINT STANDARDIZE OF "XXXTT"
TTXXX
←
```

Now that we have STANDARDIZE, ADD is a trivial task. If we didn't care whether or not the answer was in standard form, then the elementary operation WORD would serve as a working ADD procedure. Simply putting the two compact numbers together gives a new number with exactly enough T's and X's to represent the sum of the two original numbers. But, as long as we can get any representation for this sum, we might as well get the standard form representation by using STANDARDIZE.

```
←TO ADD /X/ AND /Y/
>1Ø OUTPUT STANDARDIZE OF WORD OF
    /X/ AND /Y/
>END
←
```

ADD is so simple it is hard to think of test cases for which it might be wrong. The subprocedures we wrote to help us with ADD, T'S, X'S, and STANDARDIZE, turn out to be so powerful that our original problem, ADD, has become a triviality. This is a common occurrence in mathematics. It also often happens that tools (subprocedures, subtheorems, techniques) invented to attack some problem end up having an importance of their own, quite apart from the problem they were invented for.

3.7 Comparing Compact Numbers

Before we write a procedure to compare two compact numbers, let's assume that from now on compact numbers will be written in standard form, unless stated otherwise. Thus, the inputs to compact number procedures will be in standard form and the procedure must output numbers in standard form. This requirement for the outputs is obviously desirable. The requirement that the inputs be in standard form will make our procedures less powerful (they won't handle nonstandard inputs) but simpler. If, for any reason, we decide later that we want the extra power, we can convert each nonstandard input into standard form simply by adding the instruction line

```
MAKE
  NAME:  "INPUT"
  THING:  STANDARDIZE OF /INPUT/
```

at the beginning of the procedure (with "INPUT" replaced by "X" or "M" or "FIRST" or whatever name the procedure uses for the input). These lines will put the inputs into standard form and then we won't have to worry about them again.

In comparing two compact numbers, the first thing to look at is the T's. If one of the numbers has more T's than the other, that one must be the larger. This is because, with the numbers in standard form the difference of the X's can't be more than nine; not enough to make up for an extra T. Only if the number of T's is the same for both do we need to look at the X's. In that case, of course, the larger number is simply the one with the more X's.

Comparing the number of T's (or X's) in the two numbers is just a matter of comparing the number of letters in two words. But, that is a problem we solved when we wrote the procedures to compare mark numbers. One of the programs was

```
←TO MARK-COMPARE /FIRST/ AND /SECOND/   (We changed the name to
                                         MARK-COMPARE so that we
                                         can use the name COMPARE
                                         for the compact number
                                         program)

>1Ø TEST IS /FIRST/ /SECOND/
>2Ø IF TRUE OUTPUT "EQUAL"
>3Ø TEST EMPTYP OF /FIRST/
>4Ø IF TRUE OUTPUT "SECOND"
>5Ø TEST EMPTYP OF /SECOND/
>6Ø IF TRUE OUTPUT "FIRST"
>7Ø OUTPUT MARK-COMPARE OF (BUTFIRST
    OF /FIRST/) AND (BUTFIRST OF
    /SECOND/)
>END
←
```

When we wrote MARK-COMPARE (we called it COMPARE then), we intended that it should work for words made up entirely of X's. But, there is nothing in the program that takes the least notice of what letters make up /FIRST/ and /SECOND/. So MARK-COMPARE will work with any two words, outputting the name of the word that has the larger number of letters.

To write COMPARE (for compact numbers), we'll use MARK-COMPARE to compare the T's. If these aren't EQUAL, we're done and the larger number is simply the one with the more T's. If the T's are EQUAL, then the answer to the problem is just MARK-COMPARE of the X's.

```
←TO COMPARE /FIRST/ AND /SECOND/   (/FIRST/ and /SECOND/ will
                                    always be compact numbers in
                                    standard form)
>1Ø TEST IS (MARK-COMPARE OF T'S   (Do both inputs have the same
    OF /FIRST/ AND T'S OF           number of T's?)
    /SECOND/) "EQUAL"
>2Ø IF FALSE OUTPUT MARK-COMPARE   (If they don't, the one with
    OF (T'S OF /FIRST/) AND         the more T's is the larger)
    (T'S OF /SECOND/)
```

(continued)

```
>3Ø OUTPUT MARK-COMPARE OF          (If they do have the same number
    (X'S OF /FIRST/) AND (X'S       of T's, then the answer is the
    OF /SECOND/)                    one with the more X's.  Notice
                                    that if the numbers have equal
                                    T's and equal X's, this line
                                    will output "EQUAL")

>END
←
```

We'll test COMPARE with some representative inputs.

```
←PRINT COMPARE OF "TTX" AND "XXXX"
FIRST
←PRINT COMPARE OF "TTXX" AND "TTXXX"
SECOND
←PRINT COMPARE OF "TXX" AND "TXX"
EQUAL
←
```

We were quite specific in insisting that COMPARE would be guaranteed to work only on numbers in standard form. In fact, COMPARE will work properly with many, but not all, nonstandard compact numbers. Disordering presents no problem because T'S and X'S handle that satisfactorily. Incomplete compacting (more than nine X's) is what can cause problems. (Find an example for which COMPARE gives the wrong answer.)

3.8 Subtraction of Compact Numbers

This COMPARE procedure cannot be converted into a SUBTRACT program as easily as in the mark number case. An attempt to do so would go like this. We'll make the difference between two compact numbers a new compact number whose T's equal the difference in the T's of the two original numbers and whose X's equal the difference of the X's. To do this we'll use MARK-DIFFERENCE on the T's and X's separately. Then DIFFERENCE would be -

```
←TO DIFFERENCE /F/ AND /S/
>1Ø MAKE
      NAME:  "T'S OF ANSWER"      (Find the number of T's in the
      THING:  MARK-DIFFERENCE OF   answer)
              (T'S OF /F/) AND
              (T'S OF /S/)
>2Ø MAKE
      NAME:  "X'S OF ANSWER"      (Find the number of X's in the
      THING:  MARK-DIFFERENCE OF   answer)
              (X'S OF /F/) AND
              (X'S OF /S/)
>3Ø OUTPUT WORD OF /T'S OF        (Put the T's and X's together
   ANSWER/ AND /X'S OF ANSWER/     with WORD and output)
>END
←
```

Recall the mark number subtraction procedure, MARK-DIFFERENCE:

```
←TO MARK-DIFFERENCE /M/ AND /N/
>1Ø TEST EMPTYP OF /M/
>2Ø IF TRUE OUTPUT /N/
>3Ø TEST EMPTYP OF /N/
>4Ø IF TRUE OUTPUT /M/
>5Ø OUTPUT DIFFERENCE OF (BUTFIRST
   OF /M/) AND (BUTFIRST OF /N/)
>END
←
```

If we try out DIFFERENCE, we will find that it doesn't work in all cases.

```
←PRINT DIFFERENCE OF "TTX" AND "TX"
T
←PRINT DIFFERENCE OF "TXX" AND "TTTXXXX"
TTXX
←PRINT DIFFERENCE OF "TTX" AND "TXXXX"
TXXX
←
```

Here is a bug. The correct answer is XXXXXXX, not TXXX. It's clear how the error occurred. The first input has two T's and the second has one so the difference in the T's is T. The first

input has one X and the second four, so the difference in the X's is XXX. So, DIFFERENCE output TXXX. The problem is that the smaller number has more X's than the larger number. How can we handle this?

A solution to our problem is to partially uncompact the larger number, changing one of its T's to ten X's. Then it will certainly have more X's than the smaller number (which can have no more than nine, being in standard form). When we try this on the problem that gave us the trouble, DIFFERENCE OF "TTX" AND "TXXXX" we change it to DIFFERENCE OF "TXXXXXXXXXX" AND "TXXXX". The procedure DIFFERENCE will have no difficulty with this problem, the difference of the T's is /EMPTY/ and the difference of the X's is XXXXXXX. So, the answer is XXXXXXX.

If we try to fix DIFFERENCE this way, we'll have a little trouble because we'll have to first find out which of the two input numbers is larger before we can decide whether we need to do the uncompacting. This isn't particularly difficult to do -- we already have the COMPARE procedure we need. And, since DIFFERENCE is becoming somewhat complicated, instead of repairing it, we'll write a new procedure, SUBTRACT, which uses it as a subprocedure. SUBTRACT will be like DIFFERENCE except that it will require that the first of the two inputs be larger than the second.

```
←TO SUBTRACT /BIG/ AND /SMALL/
>1Ø TEST IS (COMPARE OF X'S OF /BIG/ (Which number has more X's?)
    AND X'S OF /SMALL/) "SECOND"
>2Ø IF TRUE MAKE                     (/SMALL/ has more X's, so
      NAME:  "BIG"                    change one of the T's in
      THING:  WORD OF (BUTFIRST OF    /BIG/ to ten X's)
               T'S OF /BIG/) AND
              (WORD OF X'S OF /BIG/
              AND "XXXXXXXXXX")
>3Ø OUTPUT DIFFERENCE OF /BIG/ AND   (We've made sure that this is
    /SMALL/                           a case that DIFFERENCE can
>END                                  handle so we use it)
←
```

There are a few questions that need to be settled before we can have confidence in SUBTRACT. For example: how can we be sure, in line 2Ø, that /BIG/ has a T at all? If /BIG/ has no T, then, of course, we can't uncompact it. What line 2Ø does, in that case, is replace /BIG/ by another number, ten larger. That would be certain to foul up our subtraction. Fortunately, line 2Ø is carried out only if /SMALL/ has more X's than /BIG/ (because of the IF TRUE condition following the test in line 1Ø) and, if that is so, /BIG/ must have at least one T in order to be bigger than /SMALL/.

It is also important, if we do perform line 2Ø, that /BIG/ originally have more T's than /SMALL/ has. Otherwise, after line 2Ø is carried out, /SMALL/ will have more T's than /BIG/ and so the subprocedure DIFFERENCE won't work. (For example, consider DIFFERENCE OF "TXXXXXXXXXXXXX" AND "TTX". The correct answer is XX but DIFFERENCE will give TXXXXXXXXXXXX.) But, this cannot occur. The argument is as follows. Since /BIG/ must be larger than /SMALL/, it must start with at least the same number of T's. But, if it had exactly the same number of T's as /SMALL/, it cannot have had fewer X's. Thus we would not have performed line 2Ø. So, since we did perform line 2Ø, /BIG/ must have had more T's than /SMALL/.

Finally, we must be sure that SUBTRACT outputs in standard form, since we decided that all of our compact number procedures should deal with numbers in standard form. The output of DIFFERENCE will always have the T's to the left of the X's so the ordering requirement is satisfied. What isn't obvious is that the answer will always have fewer than ten X's. The original /BIG/ and /SMALL/ each have fewer than ten X's (since they are in standard form) so, if line 2Ø isn't used, we're all right. (The difference

between two numbers both less than ten is less than ten.) But, if line 2Ø is used, then /BIG/ will have at least ten X's, often more. How can we be sure that the answer will have fewer than ten? If we were to take away the number of X's that /BIG/ had originally, then the answer would have exactly ten X's. If we take more than that, the answer will have fewer than ten X's. But the only time line 2Ø happens is when the number of X's in /SMALL/ (that is, the number we're subtracting) *is* larger than the original number of X's in /BIG/. So, the answer will have fewer than ten X's.

Choose some representative cases and test SUBTRACT. Notice that it will do *something*, if the inputs are in the wrong order, but the result will be incorrect.

The old MARK-SUBTRACT program that we discussed previously can be made to work with compact numbers by making a very minor change. Line 4Ø was

```
4Ø MAKE
     NAME:  "COUNTER"
     THING:  WORD OF /COUNTER/ AND "X"
```

We'll change that to

```
4Ø MAKE
     NAME:  "COUNTER"
     THING:  ADD OF /COUNTER/ AND "X"
```

This makes no difference as far as mark numbers are concerned since ADD and WORD are the same procedure for them. With compact numbers, however, ADD does more than just WORD the two numbers together.

This SUBTRACT program isn't as "good" as the one we just wrote specially for compact numbers, since it is slower in getting an answer, particularly if the answer is large. It is an interesting

program though, because it depends very little on what method we use for writing numbers. It works equally well for mark numbers, compact numbers, and (as we'll see later) place value numbers. -- In fact, the reason it is a relatively slow program is that it doesn't take advantage of any of the special shortcuts that may be possible with each particular number notation.

As we find that we want to use larger and larger numbers we'll soon decide that compact numbers, while some help, are not enough. Three lines full of T's is easier to read or write than thirty lines full of X's but both tasks are unpleasant. Fortunately, we can improve the situation somewhat. We can introduce a new symbol, say H, and let each H stand for ten X's. Then we'll have to redefine standard form and rewrite most of our compact number procedures, but when we're finished we will have gained considerably in the size of the numbers we can deal with easily. If we want to work with still larger numbers, we can add yet another symbol standing for ten H's. This process can be repeated until we either have great difficulty remembering all the symbols we introduced or, hopefully, until we can handle any numbers that come up in our work. But, if we need to work with still larger numbers without introducing still more symbols, there are other number representations and some of these will be developed subsequently.

4. More About Mark Numbers

4.1 Multiplication of Mark Numbers

When we wrote the procedure UNCOMPACT, we were actually writing a sort of multiplication procedure. For each T we substituted ten X's. If our original number had only T's in it, the new

uncompacted number would be ten times as long. So, in a way, we have multiplied a mark number by ten. In the process we also changed the marks from T's to X's which, if we had been thinking of compact numbers would have made an important difference. But, with simple mark numbers a mark is a mark (that is, all marks mean the same thing). Nevertheless, let's rewrite UNCOMPACT so that it will take a word of X's (a normal mark number) and substitute ten X's for each X. Since the purpose of this program is different from UNCOMPACT, we'll give it a new name, TIMESTEN.

```
←TO TIMESTEN /N/
>1Ø TEST EMPTYP OF /N/          (We are using the second UNCOMPACT
                                 procedure as our model - the one
                                 that works on disordered compact
                                 numbers)
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF (TIMESTEN    (We don't want to check whether
    OF BUTFIRST OF /N/) AND      FIRST OF /N/ is "T". Every mark
    "XXXXXXXXXX"                 in /N/ should have "XXXXXXXXXX"
                                 substituted for it)
>END
←
```

Notice that TIMESTEN is a recursive procedure. The program says that if /N/ isn't /EMPTY/, then TIMESTEN OF /N/ is the same as TIMESTEN OF (BUTFIRST OF /N/) with ten extra X's tacked onto the end (that is, substituted for the one X that BUTFIRST chops off at the beginning).

What we are really after is a general multiplication procedure, not just one that multiplies by ten. We want a procedure that will take two inputs instead of one and will output the product of these inputs. In order to generalize TIMESTEN in this way, we first look carefully at the procedure to see where the fact that we are using <u>ten</u> rather than some other number shows up. The only place in TIMESTEN that this happens is at the end of line 3Ø

where we have "XXXXXXXXXX". If that word were changed, say to "XXXX", TIMESTEN would multiply by four. So, to make TIMESTEN multiply by its second input, we will just put the name of that input in line 3Ø.

```
←TO TIMES /N/ AND /Y/                (Let's change the name of the
                                      procedure since we won't be
                                      multiplying only by ten)

>1Ø TEST EMPTYP /N/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF (TIMES OF         (Instead of tacking on ten X's,
    BUTFIRST OF /N/ AND /Y/)          we will use the number given in
    AND /Y/                           /Y/)
>END
←
```

Let's TRACE TIMES and watch it run.

```
←TRACE TIMES
←PRINT TIMES OF "XXX" AND "XXXX"
TIMES OF "XXX" AND "XXXX"
  TIMES OF "XX" AND "XXXX"
    TIMES OF "X" AND "XXXX"
      TIMES OF "" AND "XXXX"
      TIMES OUTPUTS ""
    TIMES OUTPUTS "XXXX"
  TIMES OUTPUTS "XXXXXXXX"
TIMES OUTPUTS "XXXXXXXXXXXX"
XXXXXXXXXXXX
←
```

In order to calculate TIMES OF "XXX" AND "XXXX" the computer had to first find TIMES OF "XX" AND "XXXX" and TIMES OF "X" AND "XXXX" and TIMES OF "" AND "XXXX". So the procedure TIMES is actually called four times. Now suppose we try TIMES OF "XXXX" AND "XXX". The answer will be the same as before, XXXXXXXXXXXX, (unless there is a bug in our program) but the process of getting that answer will be different.

```
←PRINT TIMES OF "XXXX" AND "XXX"
TIMES OF "XXXX" AND "XXX"
  TIMES OF "XXX" AND "XXX"
    TIMES OF "XX" AND "XXX"
      TIMES OF "X" AND "XXX"
        TIMES OF "" AND "XXX"
        TIMES OUTPUTS ""
      TIMES OUTPUTS "XXX"
    TIMES OUTPUTS "XXXXXX"
  TIMES OUTPUTS "XXXXXXXXX"
TIMES OUTPUTS "XXXXXXXXXXXX"
XXXXXXXXXXXX
←
```

In this case the procedure is called five times. If we think about how different the two processes we've just watched are, it may seem surprising that they give the same final result. A very good, mathematical question is "why does this happen?". A good answer to this question, and there are many, would be a way of looking at the TIMES procedure that makes it clear that the order of the inputs doesn't change the final result. Most of the important advances in mathematics have come from discoveries of new ways to look at things which revealed hidden relations.

For the particular question we have here, why TIMES gives the same result even if the two inputs are switched around - we'll look at one possible explanation. First we'll relax slightly the restrictions on how we write mark numbers. We've always written the X's in a horizontal line, like XXXXX, but now let's permit ourselves to write in any patterns we please, for example,

```
X   X          X          X
  X      or    X    or    X     X.  Each of these
X   X          X            X X
               X
               X
```

represents the same number. Now, if we go back to the problem of TIMES "XXX" AND "XXXX", we can think of substituting "XXXX"

for each X in "XXX". But, let's write it like this

```
X  X  X
X  X  X
X  X  X
X  X  X
```

For each X in "XXX" we've substituted a vertical row of "XXXX". If we follow the same scheme for TIMES OF "XXXX" AND "XXX" and substitute a vertical row of "XXX" for each X in "XXXX" we get

```
X  X  X  X
X  X  X  X
X  X  X  X
```

. But now, if we look at both of these patterns, we can see that they are the same except for position, that is, by a simple half turn of the paper the three by four turns into the four by three, and vice-versa. So naturally they have the same number of X's and so they are the same mark number.

A property of this type of explanation is that it is good only if the reader is convinced by it. There are no objective grounds on which to judge it right or wrong, its value depends on how it is received. For this reason most teachers tend to collect as many explanations as they can so that when one doesn't "click" they can try another.

To go back to the TIMES procedure, notice that in the second half of the TRACE printout (the lines that go TIMES OUTPUTS) the computer is counting up by threes in one case and by fours in the other. (That is, three, six, nine, twelve, and four, eight, twelve.) The programs stopped when they came to twelve, the first number that they both reach. An interesting question is: if two programs (or people) start counting up by different amounts (in the case we did, by three and four), what numbers will both programs reach?

4.2 Counting, Copying, and Compact Multiplication

We wrote TIMES to work with mark numbers. We didn't think about how it would behave with other inputs. In fact, whenever we create a LOGO procedure we decide, at least implicitly, what sort of inputs are legal. The mere fact that the procedure is written in LOGO means that the inputs must be either words or sentences (LOGO words or sentences, of course, not necessarily English ones) and not something like a dog, a pound of butter, or a moment of joy. Often, as in the case of TIMES, we intentionally restrict the legal inputs still more. TIMES may have only mark numbers as inputs, other LOGO things are illegal. The word *domain* is used to describe the collection of legal inputs to a procedure. We would say that the domain of TIMES is pairs of mark numbers (pairs because there must be a first input, /N/, and a second input, /Y/).

Sometimes, when the inputs to a procedure are LOGO things that are not in the domain, the computer will stop and type an error comment; sometimes the computer will "go into an infinite loop" (that is, work and work but never output anything). Neither of these cases is particularly interesting. But sometimes the procedure will finish its work and output something. When this happens, it sometimes turns out that we've found a new use for an old tool as, for example, when we found we could use MARK-DIFFERENCE to get the difference of two *compact* numbers. This new use might be rather similar to the old one (like using scissors as tin snips), or it might be very different (like using a magnifying glass to start a fire). Either way the tool becomes more useful.

This is the case with the procedure TIMES. The first input, /N/, exists simply to have each of its letters replaced by the second

input, /Y/. So, it doesn't make any difference what those letters were originally. The only thing that matters is how many there are. Any word at all, used for the first input, can be replaced by the mark number containing as many X's as the original word had letters. So, in fact, we can extend the domain to allow the first input to be any LOGO word. That seems to give us nothing new or interesting. This appearance is misleading however. We can use this observation to write a short procedure called COUNT which will output the number of letters in its input. For example, COUNT OF "CAT" is "XXX" (three), COUNT OF "NARNIA" is "XXXXXX" (six), and so forth. The program is simple.

```
←TO COUNT /W/
>1Ø OUTPUT TIMES OF /W/ AND "X"
>END
←
```

TIMES just substitutes an X for each letter in /W/ and so outputs the mark number equal to the number of letters in /W/. Before we get back to TIMES, notice that we've discussed COUNT only with word inputs, not sentences. Think about COUNT of a sentence, remembering that FIRST of a sentence is the first word and BUTFIRST of a sentence is all except the first word.

```
←PRINT COUNT OF "APPLES PEARS PLUMS CHERRIES"
XXXX
←PRINT COUNT OF "APPLESPEARSPLUMSCHERRIES"
XXXXXXXXXXXXXXXXXXXXXXX
```

Varying the second input to TIMES is even more interesting. The second input is copied over as many times as there are marks (or letters) in the first input. Thus, for example,

```
←PRINT TIMES OF "XXX" AND "CAT"
CATCATCAT
←
```

This use of TIMES seems cute but not very useful. In fact, it is a surprisingly powerful technique that we will use later when we write programs whose outputs will be LOGO instructions. (Recall, in the procedure BUTFIRSTTEN, we used a line that contained BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF and so on. Procedures like TIMES can produce this kind of instruction.)

Right now, however, let's look at what happens when the second input is a compact number. The output of the procedure will be the compact number repeated as many times as the first input, a mark number, says. So, TIMES OF "XXXX" AND "TTX" is "TTXTTXTTXTTX". But, this is really the product of the mark number and the compact number, except that it isn't in standard form. Using this observation, we can write a multiplication for two compact numbers by uncompacting one of them, using TIMES to multiply the resultant mark number by the other compact number, and finally standardizing the result.

```
←TO MULTIPLY /CM/ AND /CN/
>1Ø OUTPUT STANDARDIZE OF TIMES OF
    (UNCOMPACT OF /CM/) AND /CN/
>END
←
```

Notice that we are careful to uncompact the first input of TIMES in line 1Ø and not the second. Doing it the other way would give a program that does something quite different from multiplication. This program for multiplying compact numbers is considerably better than the simple-minded one of converting both numbers to marks, multiplying the mark numbers, and then compacting the result. In fact, with a little bit of improving (to make sure that the smaller input to MULTIPLY is the one that gets uncompacted), this would be a very reasonable and practical multiplication

procedure. To achieve the most efficient multiplication possible, we'd have to consider also the problem of multiplying T's together (such as "TTT" times "TTTT"). With extended compact numbers (X's, T's, and H's), the problem is even more acute since "HHHHHH" times "HHHHH" is equal to about one teletype page (fifty lines) full of H's. We won't study the problem here, however, since we're going to solve it in a completely different way in the next chapter.

4.3 Division of Mark Numbers

In the same way that UNCOMPACT leads to mark multiplication, COMPACT leads to mark number division. One way the problem of division can be looked at is to ask how many groups of a given size can we make from the number we have. But this is precisely what COMPACT does for a special case, division by ten. In COMPACT we asked how many groups of ten there are in the input number. The answer was used as the number of T's in the compact representation of the number. Let's look again at COMPACT to see what will be involved in changing it to a general mark division procedure, that is a procedure for dividing a given mark number by *any* other mark number.

```
←TO COMPACT /WORD/
>1Ø TEST EMPTYP OF /WORD/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF (FIRSTTEN OF /WORD/) AND (COMPACT
    OF BUTFIRSTTEN OF /WORD/)
>END
←
```

Remember that FIRSTTEN would output "T" if /WORD/ had at least ten marks and it would output /WORD/ unchanged if /WORD/ had fewer than ten letters. If we're going to make COMPACT divide by

numbers other than ten, the first thing we'll have to do is to give the procedure two inputs. Let's call it

TO DIVIDE /WORD/ AND /GROUP/

where /GROUP/, the second input will be the size of the groups /WORD/ will be divided into.

Now before we go on we have to make a decision about what to do if the division doesn't come out evenly, that is, if /WORD/ won't divide into parts, all of exactly size /GROUP/. One way to handle this problem is to give an answer that consists of two numbers, a quotient (the number of groups of size /GROUP/), and a remainder (the number left over). This is what we'll eventually do but let's first write the program to just give the quotient and forget about the remainder. Then we will write a program to get the remainder and ignore the quotient. Finally, we'll write a short program to put these two things together.

To write QUOTIENT we'll use the same technique that worked for COMPACT, recursion. The important fact is that the QUOTIENT OF /WORD/ AND /GROUP/ is just one bigger than the QUOTIENT OF (/WORD/ with one group removed) AND /GROUP/. Thus, we can reduce the problem to quotients of smaller and smaller /WORD/'s until finally /WORD/ is so small that we can no longer take away a whole group. That is, until /WORD/ is smaller than /GROUP/. In that case, the quotient is zero (/EMPTY/) since no whole groups can be made (remember, we decided to skip worrying about the remainder). So we'll write,

```
←TO QUOTIENT /WORD/ AND /GROUP/
>1Ø TEST IS (COMPARE OF /WORD/ AND
    /GROUP/) "SECOND"              (Is /WORD/ smaller than /GROUP/?)
 2Ø IF TRUE OUTPUT /EMPTY/         (If it is, then no groups can be
                                    made so the answer is /EMPTY/)
>3Ø OUTPUT WORD OF "X" AND         (Otherwise, the answer is one
    QUOTIENT OF (DIFFERENCE OF     more than the quotient of /WORD/
    /WORD/ AND /GROUP/) AND        reduced by /GROUP/ and /GROUP/)
    /GROUP/
>END
←
```

Compare QUOTIENT with COMPACT. In line 3Ø of QUOTIENT the "X" plays the same role that FIRSTTEN OF /WORD/ does in line 3Ø of COMPACT. Also, DIFFERENCE OF /WORD/ AND /GROUP/ in QUOTIENT is analogous to BUTFIRSTTEN OF /WORD/ in COMPACT. Remember that /GROUP/ in QUOTIENT plays the role of the ten (XXXXXXXXXX) in COMPACT.

Let's run QUOTIENT with TRACE.

```
←TRACE QUOTIENT
←PRINT QUOTIENT OF "XXXXXXX" AND "XXX"
QUOTIENT OF "XXXXXXX" AND "XXX"
  QUOTIENT OF "XXXX" AND "XXX"  ("XXXX" is DIFFERENCE OF "XXXXXXX"
                                 AND "XXX")
    QUOTIENT OF "X" AND "XXX"
    QUOTIENT OUTPUTS ""          ("X" has no groups of three)
  QUOTIENT OUTPUTS "X"
QUOTIENT OUTPUTS "XX"
XX
←
```

So, seven contains two groups of three. It also contains one left over but we'll get that later. Notice the way QUOTIENT works, it counts backwards by three ("XXXXXXX", "XXXX", "X") and then forwards by one the same number of counts ("", "X", "XX"). We can think of this process as counting the number of times we

can take a group of three marks out of seven. If we had started with this idea instead of the recursive one, we would probably have written a slightly different procedure (using a GO TO LINE instruction) having the same effect.

Now we write REMAINDER so that we can finish DIVIDE. For this procedure, let's use the idea we were just discussing instead of recursion. That is, we'll take /GROUP/ away from /WORD/ until the final /WORD/ becomes smaller than /GROUP/. That amount, left over in /WORD/ is the remainder.

```
←TO REMAINDER /WORD/ AND /GROUP/
>1Ø TEST IS (COMPARE OF /WORD/      (Is /WORD/ smaller than /GROUP/?)
    AND /GROUP/) "SECOND"
>2Ø IF TRUE OUTPUT /WORD/          (If it is, then /WORD/ is the
                                    leftover amount already.  No
                                    more /GROUP/ can be taken away)
>3Ø MAKE
      NAME:  "WORD"
      THING:  DIFFERENCE OF /WORD/(Otherwise, take /GROUP/ away
              AND /GROUP.          from /WORD/)
>4Ø GO TO LINE 1Ø                  (And try again with the smaller
                                    /WORD/)
>END
←

←PRINT REMAINDER OF "XXXXXXX" AND "XXX"
X
←
```

Now we're ready to write a complete DIVIDE putting together QUOTIENT and REMAINDER. The simple and natural way is to output a sentence whose first word is the quotient and second the remainder.

```
←TO DIVIDE /WORD/ AND /GROUP/
>1Ø OUTPUT SENTENCE OF
    (QUOTIENT OF /WORD/ AND /GROUP/) AND
    (REMAINDER OF /WORD/ AND /GROUP/)
>END
←

←PRINT DIVIDE OF "XXXXXXX" "XXX"
XX X
←
```

So DIVIDE seems to be the whole thing. Unfortunately, it has one very annoying bug.

```
←PRINT DIVIDE OF "XXXXXXXXXXXX" AND "XXXX"
XXX
←PRINT DIVIDE OF "XXX" AND "XXXX"
XXX
```

In the first case the XXX is the quotient and there is no remainder. In the second case the XXX is the remainder and there is no quotient. The answer that DIVIDE gives is exactly the same in both cases because SENTENCE OF /EMPTY/ AND "XXX" is the same as SENTENCE OF "XXX" AND /EMPTY/. What we need is some word to hold the place of quotient as first word in the sentence even when there is no quotient. This special word would do the same for remainder. The idea of this sort of a place holder (a symbol to stand for nothing) is the great invention of Hindu mathematicians before 876 A.D. Instead of solving this problem with compact numbers by inventing a place holder, let's go on to the next section, on place numbers, where this concept will be of central importance.

5. Place Numbers

5.1 Positional Notation

When God created the integers* he probably invented them as some form of mark numbers. Mark numbers are elemental in concept. The algorithms of mark number arithmetic are beautifully direct and simple. Thus, to get the next number, tack on an x; to add two numbers, stick them together; to multiply two numbers, replace each mark in one number by the other number; and so on. This simplicity does not hold for all operations we wish to do with mark numbers, however. Some operational difficulties become evident when we try to work with large, or even moderate-sized. numbers. A number such as XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-XXXX is not only difficult to take in at a glance, but takes a long time to say, to copy, or to remember.

To more reasonably deal with larger numbers such as this, we developed a new notational scheme, the compact number representation. We decided to rewrite groups of ten X's as T, and groups of ten T's as H. Thus, the above mark number was more conveniently written TTTTX. This gain in ease of handling large numbers was accompanied by some loss in the simplicity and uniformity of the algorithms for the standard arithmetic operations. The procedures for compact number arithmetic have to deal with three kinds of marks, instead of just one as with mark numbers. Overall, though, the trade-off seems worthwhile; compact numbers are an improvement on mark numbers, at least for "large" numbers.

But what if the numbers get still larger? Consider, for example, HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHTXX. This compact number is as hard to manage as the mark number we looked at earlier.

---

*According to Dedekind, a famous mathematician, (1831-1916) God invented the positive integers, all else is the work of man.

When we consider such large numbers, the improvement over mark numbers which motivated us to devise compact numbers appears illusory. What can we do to improve this situation?

One possible solution is to further extend the compact number notation by inventing a new character whenever the numbers we want to deal with get inconveniently large. We could, for example, let K stand for ten H's and rewrite the above number as KKKKHHTXX. For even larger numbers, groups of ten K's could be written as M's, and the process continued indefinitely until we run out of symbols for new groups. This solution is feasible in concept but disadvantageous in application. People have difficulty in remembering the meanings of the individual symbols (X, T, H, K, M, and so forth) when there are very many of them. Also, the algorithms for adding, multiplying, and comparing numbers become progressively more complex as new symbols are introduced. Even with a relatively small number of symbols (say six), a great deal of fussy work goes into scanning, interpreting, rearranging, comparing, or adding compact numbers, separately for each particular symbol group. Is it possible to invent a number system where large numbers are easy to handle and also where (as with mark numbers) the algorithms for doing arithmetic are independent of the magnitude of the numbers?

The answer is yes, and the solution comes from making an appropriate modification of compact number representation. Most of the drawbacks we found with compact numbers came from using so many different marks or symbols. A new symbol, having a value of ten of the previously-largest one, was introduced each time we wished to extend the effective range of easily manipulable numbers. But, we can achieve this extension just as easily by the use of a single symbol. All we need to do is find a way to

make the same symbol function sometimes as an X, sometimes as ten X's (that is, the compact symbol T), sometimes as one hundred X's (that is, the compact symbol H), and so on indefinitely. We can accomplish this by a representation where numbers are expressed as groups of symbols, where the groups are ordered in a definite way from left to right, where spaces are used to separate adjacent groups, and where the value of a symbol depends upon which group it is in.

This description sounds very elaborate but the notion is an extremely simple one. Here is an example. The mark number XXXXXXXXXXXXXXXXXXXXXX written in compact notation as TTXXX is written in the new notation as XX XXX where the two X's in the left-most group are each worth ten of the X's in the right-most group. Similarly, the following number, in compact notation, HHHTTXXXX is written in the new notation, XXX XX XXXX where each X in the left-most group or word is worth ten of those in the middle word which, in turn, is worth ten of those in the right-most word.

Since the value of any X in this new representation depends upon which group it is in, that is its position or place, we call this notation *positional* or *place notation*, and the numbers themselves *place numbers*. In place numbers, the value of any X is ten times the value of an X in the group one place over on the right.

A minor complication with place number notation is the problem of representing empty places. For example, the compact number HHHXX, which contains no T's, cannot be written as the place number XXX XX because that is not distinguishable from the way one would write the compact number TTTXX. We could resolve this problem by using extra spaces to represent empty groups. Thus, the compact

numbers TXX, HXX, KXX, KKH, KKT could be represented, respectively, as the following place numbers.

X XX, X XX, X XX, XX X , XX X .

We would find it difficult, though, to distinguish easily between 3, 4, and 5 spaces, and when there are no X's for units in the last group (as in the last two examples), we would have to terminate place numbers with spaces and therefore would have to indicate where the last space was by putting yet another mark at the end of the number.

We shall therefore use another symbol, Ø, to stand for an empty position in a place number. Thus, in the above examples, the compact numbers shown will be represented as follows in place notation.

TXX = X XX
HXX = X Ø XX
KXX = X Ø Ø XX
KKH = XX X Ø Ø
KKT = XX Ø X Ø

5.2 Addition of Place Numbers

How can we add place numbers? A simple case, adding two one-word place numbers (say, "XXXX" and "XXXXX") is very much like adding mark numbers (except when the sum exceeds "XXXXXXXXX", since

place numbers only begin to differ from mark numbers when they exceed "XXXXXXXXX"). If, for example, the sum is "XXXXXXXXXXX", we should write it as "X XX", obtained by performing a carry from the units to the tens place. For the time being, let's ignore the problem of carrying (we'll write a program that does that later) and try to write a procedure PLACEADD that adds two place numbers, but leaves words of more than nine X's unchanged.

When we devised the algorithm for adding compact numbers, we discovered that the sum of two numbers such as "HHTXX" and "HTTTXXX" was the sum of the X's plus the sum of the T's plus the sum of the H's, or, "HHHTTTTXXXXX". More generally, the sum of two numbers is the sum of the ones, plus the sum of the tens, plus the sum of the hundreds, and so on.

This rule applies to place numbers as well as compact numbers. The sum of two place numbers such as "X XX XXX" and "XX X XX" is, first, the sum of the right-most (the LAST) words (the ones places) plus the sum of the rest (the BUTLAST) of the numbers (the tens, hundreds, thousands, etc. places). In other words, PLACEADD of "X XX XXX" and "XX X XX" is the same as the SENTENCE of PLACEADD of "X XX" and "XX X" (the BUTLASTs of the two numbers), and the sum (in this case the WORD of) "XXX" and "XX" (i.e., the LASTs of the numbers) which is "XXXXX", and goes in the units place. Continuing, PLACEADD of "X XX" and "XX X" is, in turn, the SENTENCE of PLACEADD of "X" and "XX" (i.e., the new BUTLASTs) and the sum of "XX" and "X", which is "XXX", and goes in the tens place. Finally, PLACEADD of "X" and "XX" is "XXX" (the hundreds place) and the answer is "XXX XXX XXXXX". We know we're done when one of the numbers is /EMPTY/.

Assuming that WORD is the appropriate operation for adding two words, a PLACEADD procedure might look like this.

```
←TO PLACEADD /M/ AND /N/
>1Ø TEST EMPTYP OF /M/              (The answer is /N/
>2Ø IF TRUE OUTPUT /N/               if /M/ is /EMPTY/
>3Ø TEST EMPTYP OF /N/               and /M/ if /N/ is /EMPTY/)
>4Ø IF TRUE OUTPUT /M/
>5Ø OUTPUT SENTENCE OF PLACEADD (Otherwise, the answer is obtained
    OF (BUTLAST OF /M/) AND          by PLACEADDing the BUTLASTs and
    (BUTLAST OF /N/) AND WORD        tacking the word of the LASTs
    OF (LAST OF /M/) AND             onto the right of that)
    (LAST OF /N/)
>END
←
```

When you run this PLACEADD procedure, you may find that it has some bugs. One important bug is that PLACEADD doesn't handle numbers containing Ø's in the right way. For example, the sum of "X" and "Ø" is not really "XØ", but simply "X". Similarly, the sum of "Ø" and "Ø" is ordinarily written "Ø", not "ØØ". Clearly, WORD is not quite appropriate for our purpose. We can fix this, however. Instead of using WORD in line 5Ø, let's write our own special PLACEWORD procedure to take care of the problems with handling Ø's.

```
←TO PLACEWORD /M/ AND /N/
>1Ø TEST IS /M/ Ø                   (If either number is Ø, the
>2Ø IF TRUE OUTPUT /N/               answer is the other number)
>3Ø TEST IS /N/ Ø
>4Ø IF TRUE OUTPUT /M/
>5Ø OUTPUT WORD OF /M/ AND /N/     (Otherwise, the answer is the
>END                                 WORD of the two numbers)
←
```

PLACEWORD works properly with Ø's as well as X's. Line 5Ø in PLACEADD is now changed to

```
5Ø OUTPUT SENTENCE OF PLACEADD OF (BUTLAST OF /M/) AND (BUTLAST
   OF /N/) AND PLACEWORD OF (LAST OF /M/) AND (LAST OF /N/)
```

PLACEADD still has another bug. It does not work with one-word inputs. Thus, if one calls PLACEADD as follows,

```
←PRINT PLACEADD OF "XX" AND "XXX"
```

it prints out the false result X XX XX, instead of XXXXX. This is because the procedures LAST and BUTLAST do very different things with LOGO words than they do with LOGO sentences. LAST of a sentence is the last word in that sentence and LAST of a word is the last letter or character in that word. We need /M/ and /N/ to be interpreted by PLACEADD as sentences, not words, so LAST of /M/ or LAST of /N/ will always give the whole number, not just the last character in it. A word in LOGO, say /M/, can be transformed into a one-word sentence by the command

```
MAKE
  NAME:  "M"
  THING:  SENTENCE OF /M/ AND /EMPTY/
```

The LAST of this one-word sentence /M/ will be the word /M/ itself. We need to do this with /M/ and /N/ in PLACEADD. So our final version of PLACEADD, obtained by inserting the new lines 2 and 4, looks like this:

```
←TO PLACEADD /M/ AND /N/
>2 MAKE
     NAME:  "M"                                  (Transforms the words /M/
     THING:  SENTENCE OF /M/ AND /EMPTY/          and /N/ into one-word
>4 MAKE                                           sentences)
     NAME:  "N"
     THING:  SENTENCE OF /N/ AND /EMPTY/
>1Ø TEST EMPTYP /M/                              (The rest of the procedure
>2Ø IF TRUE OUTPUT /N/                            is unchanged)
>3Ø TEST EMPTYP /N/
>4Ø IF TRUE OUTPUT /M/
>5Ø OUTPUT SENTENCE OF PLACEADD OF
    (BUTLAST OF /M/) AND (BUTLAST
    OF /N/) AND PLACEWORD OF (LAST
    OF /M/) AND (LAST OF /N/)
>END
←
```

5.3 The Place Number Successor

Before tackling the problem of writing a procedure to take care of the carrying we ignored in PLACEADD, let's get some experience with how carrying works. The ADDX procedure takes a place number as an input and outputs its successor (that is, the sum of that number and "X"). So, ADDX should work like this.

```
←PRINT ADDX "XX Ø"
XX X
←PRINT ADDX "X XXXX"
X XXXXX
←PRINT ADDX "XXXXXXXXX"
X Ø
←PRINT ADDX "X XX XXXXXXXXX
X XXX Ø
←P ADDX "XXXXXXXXX XXXXXXXXX XXXXXXXXX"
X Ø Ø Ø
```

These examples fall naturally into two cases: with or without a carry. If the input to ADDX does not have "XXXXXXXXX" in the last (right-most) word, then the successor is obtained by simply adding an "X", its the last word (the ones place), leaving the BUTLAST unaffected. Since the addition of an "X" to the ones place sometimes calls for adding "X" to "Ø", PLACEWORD has to be used for the addition, instead of just WORD, making at least part of ADDX look like this:

```
←TO ADDX /N/
>1Ø TEST IS LAST OF /N/ "XXXXXXXXX"
>2Ø IF FALSE OUTPUT SENTENCE OF (BUTLAST OF /N/) AND
    (PLACEWORD OF LAST OF /N/ AND "X")
```

This takes care of the cases without a carry. But what if the last word does contain nine X's? In the third example, ADDX OF

"XXXXXXXXX", if we were to put another X on the end, we would get "XXXXXXXXXX" (ten X's). This can be rewritten as one ten and no ones, or X Ø, and that is clearly the answer in good place number notation.

Consider the fourth example, ADDX OF "X XX XXXXXXXXX". Again there are nine X's in the ones place. Adding an X in the ones place, we decide to replace the nine X's with a Ø and to increase the number of tens in the number by one, that is by an X. How do we add an X to the number of tens in a place number? Let's determine how many tens there are in this number. The number of tens in "X XX XXXXXXXXX" is the number of X's in the tens place, plus ten times the number of X's in the hundreds place, or all together XXXXXXXXXXX. In place number notation, the number of tens in "X XX XXXXXXXXX" is therefore "X XX". This is simply the original number with the units place removed. In general, the number of tens in a place number is obtained by striking out the last place, i.e., it is the BUTLAST of the number. So to add an X to the number of tens in "X XX XXXXXXXXX", we simply add an X to the BUTLAST of the number, that is, to "X XX". Let's do this. ADDX OF "X XX" gives "X XXX". Now we can complete the problem. The successor of "X XX XXXXXXXXX" is given by SENTENCE OF (ADDX OF BUTLAST OF "X XX XXXXXXXXX") AND "Ø", that is, SENTENCE OF (ADDX OF "X XX") AND "Ø", that is, "X XXX Ø".

How about the successor of "XXXXXXXXX XXXXXXXXX XXXXXXXXX"? Finding nine X's in the ones place, we decide that the answer is the number of tens of the number increased by one "X" coupled with a "Ø" in the ones place. In other words, ADDX OF "XXXXXXXXX XXXXXXXXX XXXXXXXXX" is SENTENCE OF (ADDX OF "XXXXXXXXX XXXXXXXXX") AND "Ø". The successor of "XXXXXXXXX XXXXXXXXX" is, in turn, SENTENCE OF (ADDX OF "XXXXXXXXX") AND "Ø". Since the successor

of "XXXXXXXXX" is "X Ø", the final answer is found to be "X Ø Ø Ø".

We now can write a recursive definition of ADDX, including cases with and without carry.

```
←TO ADDX /N/
>1Ø TEST IS LAST OF /N/ "XXXXXXXXX"
>2Ø IF FALSE OUTPUT SENTENCE OF (BUTLAST OF /N/) AND
    (PLACEWORD OF LAST OF /N/ AND "X")
>3Ø OUTPUT SENTENCE OF (ADDX OF BUTLAST OF /N/) AND "Ø"
>END
←
```

The only bug in ADDX as it now stands, is the one discussed earlier with PLACEADD, the problem of one-word inputs. As in PLACEADD, the solution is to make all inputs into sentences, by adding the following line at the beginning of the procedure:

```
>5 MAKE
     NAME:  "N"
     THING:  SENTENCE OF /N/ AND /EMPTY/
```

5.4 Converting Place Numbers to Standard Form

Our current PLACEADD procedure adds two numbers, place by place, but unlike ADDX, does not carry from any place where the sum exceeds nine X's. We still have to write a place number standardization procedure that will take any place number and do the proper things to get an equivalent representation, eliminating any words which exceed XXXXXXXXX. What are these "proper things"? Let's run through an example and see if we can devise the appropriate conversion algorithm. The place number

X XXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXX

is not in standard form. Looking at the right-most place (the ones place), we find twenty-three X's, or, enough X's to increase

the number of tens by "XX" and leave three X's in the ones place. Since, as we discovered in writing ADDX, the number of tens in a place number is just the BUTLAST OF the number, the increased number of tens in our example is given by PLACEADD OF (BUTLAST OF the number) and "XX". (Let's give this the name "TENPART".) Thus, /TENPART/ is X XXXXXXXXXXXXX. To put our original non-standard number into standard form, then, we must leave "XXX" in the ones place and tack that onto the end of the standard form of /TENPART/.

So our original problem is transformed to the simpler problem of standardizing /TENPART/. How do we do that? Looking at its right-most place, we find fourteen X's, or, enough X's to increase the number of tens by "X" and leave four X's in the ones place (i.e., the tens place of the original number). So the answer to our current problem is given by PLACEADDing BUTLAST OF /TENPART/ AND "X" and then tacking "XXXX" onto the end of that (with the SENTENCE operation). Again, let's give the name "TENPART" to the new PLACEADD sum. Thus, the new /TENPART/ is "XX" and our problem is reduced to that of putting this number into standard form. Since there are less than nine X's in the right-most place of this number, no carry is necessary, and we just leave the two X's where they are. Now, because the rest of the number is /EMPTY/, our standardization is done. The result is the sentence made by successively tacking together the right-most words of each recursion, that is

XX XXXX XXX.

Looking over our work, we see that we need a procedure that will take a word of X's and determine how many groups of ten X's it contains, and another procedure to determine how many X's are left over after all these groups of ten X's are removed. Assume we

have the procedure TENS that takes a word /N/ as an input, and outputs a word of X's, one X for each full group of ten X's in /N/, and a procedure ONES that outputs the number of X's left over after all groups of ten X's have been extracted.

Using these two procedures, the standardization procedure, STANDARDIZE, follows easily.

```
←TO STANDARDIZE /N/
>1Ø MAKE
      NAME:  "N"
      THING:  SENTENCE OF /N/ AND /EMPTY/ (To take care of one-word
>2Ø TEST EMPTYP OF /N/                     inputs)
>3Ø IF TRUE OUTPUT /EMPTY/                (We are done when /N/ is
>4Ø MAKE                                   /EMPTY/)
      NAME:  "TENPART"
      THING:  PLACEADD OF (BUTLAST OF /N/) AND
              (TENS OF LAST OF /N/)
>5Ø OUTPUT SENTENCE OF (STANDARDIZE OF /TENPART/)
    AND (ONES OF LAST OF /N/)
>END
←
```

Now we need to write the two subprocedures, TENS and ONES. TENS is to output an X for any ten X's in /N/. If there are less than ten X's in /N/, then TENS outputs /EMPTY/. If there are more than nine X's, we want to add an "X" to the output, chop off ten X's from /N/, and repeat the whole process until there are no more than nine X's left. Thus, TENS of /N/ is the same as WORD OF "X" AND TENS OF (/N/ with ten X's dropped off). Assuming we have sub-sub-procedures BUTFIRSTNINE and BUTFIRSTTEN, to chop off the first nine and first ten X's in a word, respectively, then TENS is:

```
←TO TENS /N/
>1Ø TEST EMPTYP OF BUTFIRSTNINE OF /N/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF "X" AND TENS OF BUTFIRSTTEN OF /N/
>END
←
```

The procedure ONES outputs the number of X's remaining after a word has lost all the full groups of ten X's it previously contained. ONES chops off ten X's at a time until there are nine or less X's left in the word. It then outputs this remainder. ONES utilizes the sub-sub-procedures BUTFIRSTNINE and BUTFIRSTTEN also.

```
←TO ONES /N/
>1Ø TEST EMPTYP OF BURFIRSTNINE OF /N/
>2Ø IF FALSE OUTPUT ONES OF BUTFIRSTTEN OF /N/
>3Ø TEST EMPTYP OF /N/
>4Ø IF TRUE OUTPUT "Ø"              ("Ø" denotes an empty place)
>5Ø OUTPUT /N/                      (Otherwise the result is a num-
>END                                 ber between "X" and "XXXXXXXXX")
←
```

BUTFIRSTNINE and BUTFIRSTTEN are straightforward. (BF is the legal LOGO abbreviation for BUTFIRST.)

```
←TO BUTFIRSTNINE /N/
>1Ø OUTPUT BF OF BF OF BF OF BF OF BF OF BF OF BF OF BF OF
    BF OF /N/
>END
←

←TO BUTFIRSTTEN /N/
>1Ø OUTPUT BF OF BUTFIRSTNINE OF /N/
>END
←
```

With STANDARDIZE completed, we can now go back and fix PLACEADD to output answers in standard place number form.

```
←TO NEWPLACEADD /M/ AND /N/
>1Ø OUTPUT STANDARDIZE OF PLACEADD OF /M/ AND /N/
>END
←
```

Thus, NEWPLACEADD is a general procedure for adding any two place numbers and giving the answer as a place number in standard form.

5.5 Comparing Place Numbers

After the experience gained with counting and adding place numbers, it is instructive to compare numbers in place notation with equivalent mark numbers. People like to generate enormous number strings on the computer and we can help them do this through using the demonstration program EXPLODE (see appendix). EXPLODE takes a place number as an input and outputs the equivalent mark number. It is called EXPLODE because it takes a relatively short string and converts it into a relatively long one, thereby illustrating the great improvement in compression achieved with place notation over mark notation.

Now let's consider how we can compare two place numbers to see which of them is larger. There are several ways. Any of the algorithms we derived for comparing mark numbers (most of these involve counting up or down to one of the numbers or to /EMPTY/) could be adapted for use with place numbers using ADDX or writing a ONELESS procedure for getting the difference between a place number and "X". Another, indirect, way is to use EXPLODE to convert the place numbers into mark numbers and then compare the mark numbers. Thus,

```
←TO PLACECOMPARE /M/ AND /N/
>1Ø OUTPUT COMPARE OF (EXPLODE OF /M/) AND
    (EXPLODE OF /N/)
>END
←
```

However, each of the methods for comparison so far proposed is inefficient because it fails to make use of the positional information provided by place notation. For example, if two place numbers in standard form contain different numbers of words (places), we know the answer right away, since an X in any place

but the right-most represents more X's than all of the X's in all of the places to its right. For example, "X Ø Ø" is larger than "XXXXXXXXX XXXXXXXXX". Thus, in general, the longer sentence is the larger number.

If both numbers have the same number of places, and the numbers are not identical, then we will have to go through the places, one by one from left to right, comparing corresponding words until we find a larger word, and hence, the larger number. Let's assume we have a procedure MEASURE that takes two standardized place numbers and outputs "FIRST" or "SECOND" according to which number has more words or "SAME" if both numbers have the same number of words.

Let's also assume we have a procedure COMPAREWORDS that scans two unequal place numbers of the same size (i.e., with the same number of words), word by word, until two corresponding words are found to be unequal. It then outputs "FIRST", if the word from the first number is larger than the corresponding word from the second number; otherwise, it outputs "SECOND". Now the main procedure PLACECOMPARE, which compares two place numbers in standard form, is easy to write. It checks to see if the two numbers are equal; if not, it checks to see if they have the same number of places (words); if so, it compares corresponding words using COMPAREWORDS.

```
←TO PLACECOMPARE /FIRST/ AND /SECOND/
>1Ø TEST IS /FIRST/ /SECOND/          (First see if the numbers are
                                       identical)
>2Ø IF TRUE OUTPUT "SAME"
>3Ø MAKE
      NAME:  "LENGTH"                 (Find whether the two numbers
      THING:  MEASURE OF /FIRST/       have the same number of
              AND /SECOND/             places)
                                                    (continued)
```

```
>4Ø TEST IS /LENGTH/ "SAME"
>5Ø IF FALSE OUTPUT /LENGTH/      (If not, the answer is the number
                                   with the greater number of places)
>6Ø OUTPUT COMPAREWORDS OF        (If so, compare the individual
    /FIRST/ AND /SECOND/           places to find which number is
>END                               larger)
←
```

MEASURE works as follows. It peels a word off each number and tests to see whether one or both numbers have become /EMPTY/. If not, the process is repeated until at least one of the numbers becomes /EMPTY/. The number that becomes /EMPTY/ first is the smaller; if both numbers become /EMPTY/ at the same time, they are the same in length (number of words) though they are unequal in numerical value.

```
←TO MEASURE /FIRST/ AND /SECOND/
>1Ø TEST BOTH (EMPTYP OF /FIRST/)  (The numbers have the same
    AND (EMPTYP OF /SECOND/)        length if they become empty
>2Ø IF TRUE OUTPUT "SAME"           together)
>3Ø TEST EMPTYP OF /FIRST/         (If one of them becomes empty
>4Ø IF TRUE OUTPUT "SECOND"         first, the other is longer)
>5Ø TEST EMPTYP OF /SECOND/
>6Ø IF TRUE OUTPUT "FIRST"
>7Ø OUTPUT MEASURE OF (BUTFIRST OF (Continue the process with the
    /FIRST/) AND (BUTFIRST OF       two reduced numbers)
    /SECOND/)
>END
←
```

The other subprocedure in PLACECOMPARE is COMPAREWORDS. This procedure compares corresponding words in the two numbers which at this point are known to be unequal and to contain the same number of words. The numbers will differ in at least one of the places. The left-most place in which they differ will determine which is larger. So, the procedure compares corresponding places from left to right.

For example, it compares the two numbers, "XX XXX XX" and "XX XXXX X", as follows. It begins by checking the FIRST word of each number. Since these are equal, it determines that the answer to the problem is the same as the answer obtained by comparing the BUTFIRST's of the numbers. So it compares "XXX XX" and "XX XXXX". The left-most (FIRST) places differ. So the one which has a larger value in that place is the larger number. To decide which of "XXX" and "XX" is larger, the procedure first checks to see if either of these places is Ø. If so, i.e., if this is a Ø place for one of the numbers, the other number is larger. If not, as in the example, one of the three mark number comparison procedures (all of which were called COMPARE - see Section 2.3) is invoked to determine which place, and therefore which number, is larger. (In the present case it would find the first number is larger.)

```
←TO COMPAREWORDS /FIRST/ AND /SECOND/
>1Ø TEST IS (FIRST OF /FIRST/)          (If the first words are the
    (FIRST OF /SECOND/)                  same, the numbers differ in
>2Ø IF TRUE OUTPUT COMPAREWORDS OF       some other word, so compare
    (BUTFIRST OF /FIRST/) AND            the rest of the two numbers)
    (BUTFIRST OF /SECOND/)
>3Ø TEST IS FIRST OF /FIRST/ "Ø"        (If one of the two words is
>4Ø IF TRUE OUTPUT "SECOND"              "Ø", the other number is
>5Ø TEST IS FIRST OF /SECOND/ "Ø"        the larger)
>6Ø IF TRUE OUTPUT "FIRST"
>7Ø OUTPUT COMPARE OF (FIRST OF        (If neither word is "Ø",
    /FIRST/) AND (FIRST OF               compare the two words as
    /SECOND/)                            mark numbers)
>END
←
```

We have assumed throughout that the two place numbers being compared are in standard form. If not, PLACECOMPARE will not work. (STANDARDIZE can, of course, be used to turn any place number into standard form.) We further assume that standard form means the numbers are sentences so the problem of one-word, non-sentence inputs does not arise.

5.6 Subtraction of Place Numbers

While discussing PLACECOMPARE, we mentioned that by using ADDX or ONELESS we could have written COMPAREs for place numbers that were analogous to the mark number COMPAREs. For example, this procedure to compare mark numbers,

```
←TO COMPARE /FIRST/ AND /SECOND/
>1Ø TEST IS /FIRST/ /SECOND/
>2Ø IF TRUE OUTPUT "SAME"
>3Ø MAKE
      NAME:  "TEMP"
      THING:  /EMPTY/
>4Ø TEST IS /TEMP/ /FIRST/
>5Ø IF TRUE OUTPUT "SECOND"
>6Ø TEST IS /TEMP/ /SECOND/
>7Ø IF TRUE OUTPUT "FIRST"
>8Ø MAKE
      NAME:  "TEMP"
      THING:  WORD OF /TEMP/ AND "X"
>9Ø GO TO LINE 4Ø
>END
←
```

becomes a PLACECOMPARE if line 8Ø is rewritten as

```
>8Ø MAKE
      NAME:  "TEMP"
      THING:  ADDX OF /TEMP/
```

Similarly, a PLACEADD can be written that simply counts up to the sum using ADDX, as follows. Suppose /M/ and /N/ are place numbers, and a third one /TEMP/ is "Ø". Now, if /TEMP/ is the same as /M/ (i.e., /M/=/TEMP/=Ø), then the sum of /M/ and /N/ is simply /N/. If /TEMP/ is not /M/, then perhaps /TEMP/+"X" will be. If /TEMP+"X" is /M/, then the answer is /N/+"X". If not, then we can try /TEMP/+"X"+"X", and so on. In other words, if /TEMP/ and /N/ are both increased one "X" at a time (using ADDX), when /TEMP/ finally does equal /M/, /N/ will contain the answer.

```
←TO PLACEADD /M/ AND /N/
>1Ø MAKE
      NAME:  "TEMP"
      THING:  "Ø"
>2Ø TEST IS /TEMP/ /M/
>3Ø IF TRUE OUTPUT /N/
>4Ø MAKE
      NAME:  "TEMP"
      THING:  ADDX OF /TEMP/
>5Ø MAKE
      NAME:  "N"
      THING:  ADDX OF /N/
>6Ø GO TO LINE 2Ø
>END
←
```

Along the same lines, we can write a procedure, PLACESUB, that outputs the difference of two place numbers, using ADDX. One such procedure,

```
←TO PLACESUB /M/ AND /N/
>1Ø TEST IS /M/ /N/
>2Ø IF TRUE OUTPUT "Ø"
>3Ø OUTPUT ADDX OF (PLACESUB OF /M/) AND
    (ADDX OF /N/)
>END
←
```

is based on the principle that the difference of, say, "X XX" and "X Ø" is one more than the difference of "X XX" and "X X", which is one more than the difference of "X XX" and "X XX" which is "Ø". Procedures like these are easy to write but they get very slow when their input numbers get big. In order to PLACESUB "XXX X XXX" AND "X", for example, we have to wait for ADDX to count to "XXX X XX" twice. As with COMPARE and PLACEADD, however, more efficient algorithms can be devised.

Consider the two place numbers

XXXX XX XXX and XX X X.

The difference between them is the number that would have to be added to the second to get the first. Now, since the sum of two place numbers (unless there is a carry) is the sum of the ones, plus the sum of the tens, plus the sum of the hundreds, and so on, the difference of two place numbers is the number of ones that must be added to the ones place of the second in order to get the number of X's in the ones place of the first plus the number of tens needed, plus the number of hundreds, and so on. The difference of two place numbers, then, is the sentence of the differences of each of the places. To determine the difference of "XXXX XX XXX" and "XX X X", we first find the difference of the ones places (LAST words) which is "XX", and put that at the end of the difference of the remaining parts (the BUTLASTs) of the numbers, that is

"XXXX XX" and "XX X".

That difference, however, is just the SENTENCE of (the difference of "XXXX" and "XX") and "X". The difference of "XXXX" and "XX" is "XX", so our answer, the sentence made from all the partial differences is "XX X XX". Assuming we have a procedure UNWORD* that, given two place words /M/ and /N/, where /M/ is greater than or equal to /N/, outputs their difference, a PLACESUB which performs the process just described can be written as follows.

```
←TO PLACESUB /M/ AND /N/
>1Ø MAKE
      NAME:  "M"
      THING:  SENTENCE OF /M/ AND /EMPTY/
>2Ø MAKE
      NAME:  "N"
      THING:  SENTENCE OF /N/ AND /EMPTY/
>3Ø TEST EITHER (IS /M/ /EMPTY/) OR (IS /M/ "Ø")
>4Ø IF TRUE OUTPUT /N/
>5Ø OUTPUT SENTENCE OF PLACESUB OF (BUTLAST OF /M/) AND
    (BUTLAST OF /N/) AND UNWORD OF (LAST OF /M/) AND
    (LAST OF /N/)
>END
←
```

---

*The name UNWORD is appropriate. UNWORD is the opposite of WORD, which finds the *sum* of two place words, i.e., mark numbers.

This PLACESUB, however, will not work for a large class of subtraction problems, cases involving "borrowing". For example, since the sum of "XXXXXX XXXXXX" and "XXXX XXXXXX" is "X X XX", the difference of "X X XX" and "XXXXXX XXXXXX" must be "XXXX XXXXXX". How would PLACESUB find the difference of "X X XX" and "XXXXXX XXXXXX"? In attempting to find the difference between the one's places, it would send "XX" and "XXXXXX" off to UNWORD. UNWORD, however, is undefined if its first input is smaller than the second, so we're in trouble. How, in general, does one find the difference of two place numbers if a particular place in the larger number is smaller than the corresponding place in the smaller number?

When subtracting compact numbers, we ran into a similar problem. Trying to find the difference between, for example, "TTTXXX" and "TXXXXX", we found we couldn't find the difference between the X's without first converting one of the T's in the first number into the ten X's it represents. The problem then became the difference of "TTXXXXXXXXXXXX" and "TXXXXX". This problem can be solved by finding the difference of the X's and tacking that onto the end of the difference of the T's.

A similar algorithm can be applied to place numbers. If, at any time in PLACESUB we find the LAST OF /N/ greater than the LAST OF /M/, then we will have to convert one of the X's from the next place on the left to ten X's and add them to the LAST OF /M/. The solution, then, is the sentence of the difference of the diminished (by "X") BUTLAST OF /M/ and BUTLAST OF /N/, and the UNWORD OF the WORD of "XXXXXXXXXX" and LAST OF /M/ and LAST OF /N/. Assuming we have a procedure ONELESS which outputs the difference of its input and "X", PLACESUB can now be written

```
←TO PLACESUB /M/ AND /N/
>1Ø MAKE
      NAME:  "M"
      THING:  SENTENCE OF /M/ AND /EMPTY/
>2Ø MAKE
      NAME:  "N"
      THING:  SENTENCE OF /N/ AND /EMPTY/
>3Ø TEST EMPTYP OF /N/
>4Ø IF TRUE OUTPUT /M/
>5Ø TEST IS COMPARE OF (LAST OF /M/) AND
    (LAST OF /N/) "SECOND"
>6Ø IF TRUE OUTPUT SENTENCE OF PLACESUB OF
    (BUTLAST OF /M/) AND (BUTLAST OF /N/) AND
    UNWORD OF (LAST OF /M/) AND (LAST OF /N/)
>7Ø MAKE
      NAME:  "TEMP1"
      THING:  PLACESUB OF ONELESS OF BUTLAST OF
              /M/ AND BUTLAST OF /N/
>8Ø MAKE
      NAME:  "TEMP2"
      THING:  UNWORD OF (PLACEWORD OF LAST OF /M/)
              AND "XXXXXXXXXX") AND LAST OF /N/
>9Ø OUTPUT SENTENCE OF /TEMP1/ AND /TEMP2/
>END
←
```

We now must write the subprocedures UNWORD and ONELESS. UNWORD takes two one-word inputs and outputs their difference. However, since single words of place numbers are essentially mark numbers (except for the Ø's), one of the mark subtraction procedures will do here, with the addition of the condition that if /M/ and /N/ are equal we want to output "Ø". Thus, an UNWORD can be written as follows

```
←TO UNWORD /M/ AND /N/
>1Ø TEST IS /M/ /N/
>2Ø IF TRUE OUTPUT "Ø"
>3Ø TEST EITHER (IS /M/ /EMPTY) (IS /N/ "Ø")
>4Ø IF TRUE OUTPUT /M/
>5Ø OUTPUT UNWORD OF (BUTFIRST OF /M/) AND
    (BUTFIRST OF /N/)
>END
←
```

In ONELESS we wish to output the difference of a place number and "X". Sometimes this merely involves chopping an X off the ones place of the number. However, if the ones place contains a Ø, there are no X's to chop off and we will have to convert one of the X's from the next place to the left into the ten X's it represents in the place to the right, and then chop off one of these X's. In other words, the solution will be ONELESS of the BUTLAST of the word, SENTENCEd to ten X's with one chopped off, or "XXXXXXXXX". The only other special case is when the right-most word contains only one "X", in which case we simply replace that word with a "Ø".

```
←TO ONELESS /N/
>1Ø TEST IS LAST OF /N/ "Ø"
>2Ø IF TRUE OUTPUT SENTENCE OF ONELESS OF
    (BUTLAST OF /N/) AND "XXXXXXXXX"
>3Ø TEST IS LAST OF /N/ "X"
>4Ø IF TRUE OUTPUT SENTENCE OF (BUTLAST OF
    /N/) AND "Ø"
>5Ø OUTPUT SENTENCE OF (BUTLAST OF /N/)
    AND (BUTLAST OF LAST OF /N/)
>END
←
```

Note that in the standard case (line 5Ø), the X is chopped off the last word of /N/ by the operation BUTLAST OF LAST OF /N/.

6. Modern Numbers

6.1 Digits

We invented place number notation to obtain an economy of compression in representing large numbers and a set of algorithms for arithmetic that worked uniformly for all numbers. We were successful in both respects. Of course, there were some costs. For one thing, the algorithms are more complex (though generally more efficient in computational speed). Also, compared to compact numbers, place numbers often involve long strings of words of many X's, which demand a good deal of effort to copy, and can be quite difficult to evaluate. Thus, without actually counting the X's, you probably cannot tell which of the two numbers "XXXXXXXX XXXXXXXXX" "XXXXXXXXX XXXXXXXX" is larger.

A straightforward modification of place numbers provides some help with these problems and yields the familiar representation called Arabic numbers or radix numbers or often, simply, numbers (we will call them *modern numbers* or *mod numbers* for short). The innovation is the introduction of a set of symbols to represent each of the mark numbers from /EMPTY/ or Ø through "XXXXXXXXX". These are listed in the following table showing the modern number symbols alongside the equivalent mark numbers.

Mark		Modern
/EMPTY/ or Ø	-	Ø
X	-	1
XX	-	2
XXX	-	3
XXXX	-	4
XXXXX	-	5
XXXXXX	-	6
XXXXXXX	-	7
XXXXXXXX	-	8
XXXXXXXXX	-	9

These symbols, or "digits", can be put together exactly as place numbers, with each place representing ten times the previous place. (For the time being we'll retain the empty space to separate successive places.) For example, the place number

"XXXXXXXX XXXXXXXXX"

becomes, in modern notation,

8 9

where the "8" represents eight tens and the "9" nine ones. Not only is this a more compact notation (in this case reducing 17 X's to two symbols), but once the meaning of the ten-digit symbols is acquired, these numbers are considerably easier to evaluate and to compare than any of the previous ones. The comparison we made above, between the two place numbers "XXXXXXXX XXXXXXXXX" and "XXXXXXXXX XXXXXXXX", is enormously easier with the equivalent modern numbers: 8 9 and 9 8.

Note that any set of ten distinct symbols could have been used instead of the ones we chose, which are the particular characters standardly used. The choice of *ten* as the number of symbols is equally arbitrary -- as an historical accident, corresponding perhaps to the number of fingers usually available for counting. If we had made each word of a place number represent, say, *twelve* X's in the previous place, then clearly we would have needed twelve different symbols to represent all the strings of X's that could occur in a word.

The mark-modern conversion table provides us with a single character to stand for each of the words (Ø → XXXXXXXXX) that can occur within a standard-form place number. Thus a modern number in standard form will have exactly one digit in each place. Note further that, unlike the strings of X's which, without spaces to separate the places, lose their integrity (e.g., XXX XX → XXXXX), these ten digits are easily distinguishable with or without spaces (3 2 → 32).

Since we do not wish to clutter our notation with unnecessary spaces, we shall express modern numbers as words rather than sentences, i.e., as strings of characters from the set {Ø,1,2,3,4,5,6,7,8,9} written without spaces and ordered in such a way that each character represents a place and each place but the right-most represents ten times the value of the place to its immediate right. Thus, the modern number "6453" represents six thousands, four hundreds, five tens, and three ones. So defined, there is no such thing as a non-standard modern number, since (except for zero) any number that can be expressed in modern notation can be written in only one way, and every word of digits (including strings of Ø's) that can be written represents a number.

6.2 Conversion Between Mod Numbers and Place Numbers

It is useful at this point to write procedures that perform the conversions from place numbers to modern and vice versa. We will write a procedure MODERN which takes as an input a place number, say "XX XXXXXXXX XXX" and outputs its modern equivalent "283". If we had a procedure, MOD, that converted a word of X's to its digital equivalent, the MODERN of a place number, say "XXX XXXXX", would be the same as the MOD OF FIRST of the number, or "3", with the MODERN of the rest of the number, "XXXXX", tacked on. MOD OF "XXXXX" is, in turn, "5", and, since the rest of the number is /EMPTY/, MODERN should output "35". Thus, MODERN will convert /N/, one word at a time from left to right until /N/ becomes /EMPTY/.

```
←TO MODERN /N/
>1Ø MAKE "N" (SENTENCE OF /N/ AND /EMPTY/) (To fix one-word inputs)
>2Ø TEST EMPTYP OF /N/
>3Ø IF TRUE OUTPUT /EMPTY/
>4Ø OUTPUT WORD OF (MOD OF FIRST OF /N/)
    AND (MODERN OF BUTFIRST OF /N/)
>END
←
```

The procedure for converting single words, MOD, will be simpler if we use a table in which strings of marks are matched to their equivalent digits. Then we can just "look-up" the proper number of X's to find the modern equivalent. An easy way to do this is to actually use the various strings of X's as NAMEs whose THINGs are the associated digits. This can be accomplished with a series of MAKE commands, as in the following procedure.

```
←TO MOD-TABLE
>1Ø MAKE
      NAME:  "X"
      THING:  "1"
>2Ø MAKE "XX" "2"          (This is shorthand for MAKE
                                                NAME:  "XX"
                                                THING:  "2"
>3Ø MAKE "XXX" "3"         (Similarly for these other commands)
      :    :    :
>8Ø MAKE "XXXXXXXX" "8"
>9Ø MAKE "XXXXXXXXX" "9"
>1ØØ MAKE "Ø" "X"
>END
←
```

We will perform this procedure, thereby assigning the list of mark numbers as names for their corresponding symbols. Thus, the MOD of any word of a place number will simply be the THING of that word, for example, THING OF "XXXXXX" outputs "6". (The LOGO operation THING takes an input, interprets it as a NAME and

then outputs the corresponding THING, if there is one; otherwise, it outputs /EMPTY/.) Thus, line 4Ø in MODERN can now be rewritten

```
>4Ø OUTPUT WORD OF (THING OF FIRST OF /N/) AND (MODERN
    OF BUTFIRST OF /N/)
```

The procedure, PLACE, which converts modern numbers into place numbers, can be written in a way quite analogous to MODERN, if we set up a conversion table using the following procedure

```
←TO PLACE-TABLE
>1Ø MAKE "1" "X"
>2Ø MAKE "2" "XX"
      :   :   :
>8Ø MAKE "8" "XXXXXXXX"
>9Ø MAKE "9" "XXXXXXXXX"
>1ØØ MAKE "Ø" "Ø"
>END
←
```

and then perform it. Then PLACE can be written as follows,

```
←TO PLACE /N/
>1Ø MAKE "N" (SENTENCE OF /N/ AND /EMPTY/)
>2Ø TEST EMPTYP OF /N/
>3Ø IF TRUE OUTPUT /EMPTY/
>4Ø OUTPUT SENTENCE OF (THING OF FIRST OF /N/)
    AND (PLACE OF BUTFIRST OF /N/)
>END
←
```

6.3 Mod Arithmetic Along the Lines of Place Arithmetic

With PLACE and MODERN written, any of the arithmetic operations for which we wrote place number procedures can be made to work with modern numbers. For example, a successor procedure for modern numbers, NEXT, can be written by converting the input to place notation (using PLACE), finding the successor of that place

number (using ADDX), and converting that successor back into digital notation (using MODERN).

```
←TO NEXT /N/
>1Ø OUTPUT MODERN OF ADDX OF PLACE OF /N/
>END
←
```

However, since this approach doesn't teach us anything new about MODERN numbers, we will try to write our procedures without referring back to place numbers. How, without using place numbers, can we write NEXT?

Since the places in modern numbers mean the same thing as they do in place numbers, NEXT will treat numbers ending with "9" in a special way (just as ADDX did with numbers ending with "XXXXXXXXX"). If the LAST of the numbers is not "9", then NEXT of that number is just BUTLAST of the number with the successor of the last digit tacked onto the end. If the LAST of the number does end in "9", then we need to replace the "9" with a Ø and increase the number of tens by "1". Since the number of tens in a number /N/ is BUTLAST of /N/, NEXT should output WORD OF NEXT OF BUTLAST OF /N/ AND "Ø". For example, NEXT OF "379" is the same as NEXT OF "37" with a "Ø" tacked onto the end. NEXT OF "37" is "38", so NEXT OF "379" should output "38Ø".

Assuming we have a procedure NEXT2 which outputs the successor of all one-digit inputs except "9", NEXT can now be written:

```
←TO NEXT /N/
>1Ø TEST IS LAST OF /N/ "9"
>2Ø IF TRUE OUTPUT WORD OF (NEXT OF BUTLAST OF /N/) AND "Ø"
>3Ø OUTPUT WORD OF (BUTLAST OF /N/) AND (NEXT2 OF LAST OF /N/)
>END
←
```

A quick and simple NEXT2 can be written using a table of successors obtained by performing a series of commands something like this.

```
MAKE "S" "1"        (NEXT2 OF /EMPTY/ should be "1")
MAKE "SØ" "1"
MAKE "S1" "2"
MAKE "S2" "3"
MAKE "S3" "4"
      :    :
MAKE "S8" "9"
```

Then the successor of any digit but "9" will be determined by the following procedure.

```
←TO NEXT2 /DIGIT/
>1Ø OUTPUT THING OF WORD OF "S" AND /DIGIT/
>END
←
```

We can test our NEXT procedure very quickly by running a COUNTUP procedure for modern numbers.

```
←TO COUNTUP /N/
>1Ø PRINT /N/
>2Ø COUNTUP NEXT OF /N/
>END
←
```

As we saw with compact and positional notations, most of the arithmetic operations can be defined in terms of the appropriate successor function for that notation. Similarly, once NEXT is written, it can be used to define simple (but slow) addition, comparison, and subtraction procedures for modern numbers.

An ADD procedure for digits defined in terms of NEXT can be written analogously to the PLACEADD procedure we wrote using ADDX. Assume that a counter is started off at "Ø" and increased in steps

of 1, with one of the addends, say /M/, using the procedure NEXT, until the counter equals the other addend (/N/). Then /M/ will always contain the sum of /COUNTER/ and the original /M/ and, when /COUNTER/ is /N/, /M/ will be the sum of the original /M/ and /N/. For example, when ADDing "7" and "4", /COUNTER/ grows from "Ø" to "4" while /M/ grows from "7" to the sum of "7" and "4", "11".

```
←TO ADD /M/ AND /N/
>1Ø MAKE "COUNTER" "Ø"
>2Ø TEST IS /N/ /COUNTER/
>3Ø IF TRUE OUTPUT /M/
>4Ø MAKE "COUNTER" (NEXT OF /COUNTER/)
>5Ø MAKE "M" (NEXT OF /M/)
>6Ø GO TO LINE "2Ø"
>END
←
```

A modern number COMPARE procedure defined in terms of NEXT follows closely from the algorithms for comparison of mark numbers. The following procedure compares mark numbers and outputs "FIRST", "SECOND", or "SAME" accordingly.

```
←TO COMPARE /FIRST/ AND /SECOND/
>1Ø TEST IS /FIRST/ /SECOND/
>2Ø IF TRUE OUTPUT "SAME"
>3Ø MAKE "TEMP" /EMPTY/
>4Ø TEST IS (WORD OF /TEMP/ AND /FIRST/) /SECOND/
>5Ø IF TRUE OUTPUT "SECOND"
>6Ø TEST IS (WORD OF /TEMP/ AND /SECOND/) /FIRST/
>7Ø IF TRUE OUTPUT "FIRST"
>8Ø MAKE "TEMP" (WORD OF /TEMP/ AND "X")
>END
←
```

This procedure can be easily transformed into a modern number compare by replacing the mark addition procedure (WORD) in lines 4Ø and 6Ø with a modern number add (ADD) and by replacing the mark number successor (WORD OF /TEMP/ AND "X") in line 8Ø with

the modern successor (NEXT OF /TEMP/). Through similar transformations any of the mark compares can be made to work with modern numbers.

We can write a subtraction algorithm for modern numbers following directly from the place number subtraction procedure PLACESUB that we wrote in terms of ADDX. Thus, for two modern numbers /M/ and /N/, where /N/ is less than /M/, the difference of /M/ and /N/ can be found using the following procedure.

```
←TO DIF /M/ AND /N/
>1Ø TEST IS /M/ /N/
>2Ø IF TRUE OUTPUT "Ø"
>3Ø OUTPUT NEXT OF DIF OF (/M/) AND (NEXT OF /N/)
>END
←
```

A TRACE of DIF illustrates its operation.

```
←TRACE DIF
←PRINT DIF OF "8" AND "5"
DIF OF "8" AND "5"
  DIF OF "8" AND "6"          (/N/ has been increased by 1)
    DIF OF "8" AND "7"          (and again)
      DIF OF "8" AND "8"        (and again)
      DIF OUTPUTS "Ø"           (/M/=/N/ so DIF outputs "Ø")
    DIF OUTPUTS "1"             (NEXT OF Ø)
  DIF OUTPUTS "2"               (NEXT OF 1)
DIF OUTPUTS "3"                 (NEXT OF 2)
3                               (DIF prints the final answer)
←
```

Another useful procedure that can be written in terms of NEXT is COUNT. This procedure takes as input a mark number (or, for that matter, any string of characters) and outputs its modern number equivalent. Recognizing that the COUNT OF /N/ is one more than the COUNT OF BUTLAST OF /N/ and that the COUNT OF /EMPTY/ is "Ø"; the procedure follows easily:

```
←TO COUNT /N/
>1Ø TEST EMPTYP /N/
>2Ø IF TRUE OUTPUT "Ø"
>3Ø OUTPUT NEXT OF (COUNT OF BUTLAST OF /N/)
>END
←
```

Once again we will use TRACE to clarify the operation of the procedure.

```
←TRACE COUNT
←PRINT COUNT OF "XXXXX"
COUNT OF "XXXXX"
  COUNT OF "XXXX"                (Each time through we chop an "X"
    COUNT OF "XXX"                off the input)
      COUNT OF "XX"
        COUNT OF "X"
          COUNT OF ""            (until /N/ is /EMPTY/,)
          COUNT OUTPUTS "Ø"      (and its COUNT is "Ø")
        COUNT OUTPUTS "1"        (NEXT OF "Ø" is "1")
      COUNT OUTPUTS "2"          (NEXT OF "1" is "2")
    COUNT OUTPUTS "3"                  .
  COUNT OUTPUTS "4"                    .
COUNT OUTPUTS "5"                (and finally, NEXT OF "4" is "5")
5                                (COUNT prints the final answer)
←
```

6.4 Adding Money Numbers

When we studied place number addition, we divided the problem into two parts: writing an addition procedure without carrying facilities (i.e., sometimes leaving more than nine X's in one place) and later writing a procedure that performed the carrying by putting the sum into "standard" form (i.e., leaving no more than nine X's in any place). Because modern numbers cannot be represented in non-standard form and have only one digit per place, our final addition procedure will have to do the carrying as it goes along. Before we write this procedure, however, we will design a two-part addition procedure similar to the one we used

with place numbers. (PLACEADD OF /M/ AND /N/ added two place numbers, /M/ and /N/, in standard or non-standard form. STANDARDIZE of the result put it into standard place number form.) We will carry out the new addition procedure using a familiar variant of modern numbers called *money numbers*.

A money number is a standard expression for a specified number of dollars, dimes, and cents; for example, 3 DOLLARS 4 DIMES 6 CENTS. Since ten cents equals one dime and ten dimes equal one dollar, clearly money numbers are a form of positional notation just as place or modern numbers. However, since it is perfectly meaningful to talk about, say, 22 CENTS, or 29 DIMES, money numbers, unlike modern numbers, can have non-standard representations, and a two-part addition algorithm can be written for these numbers.

Before we start adding money, though, we must have some -- so our first problem is to write a procedure, MONEY, that takes a modern number and outputs its money number equivalent in standard form. Thus, MONEY OF "294" would be "2 DOLLARS 9 DIMES 4 CENTS". One way to do this is to have three procedures, DOLLARS, DIMES, and CENTS each of which takes a modern number, say "648", as an input and outputs the two-word sentence of the contents of the proper place and its corresponding denomination (in our example, "6 HUNDREDS", "4 DIMES", and "8 CENTS", respectively). The outermost ("top-level") procedure MONEY merely puts these three two-word sentences into one big sentence of six words.

```
←TO MONEY /N/
>1Ø OUTPUT SENTENCE OF (DOLLARS OF /N/) AND SENTENCE
    OF (DIMES OF /N/) AND (CENTS OF /N/)
>END
←
```

Since the last digit in a modern number is the one's place, CENTS OF /N/ is simply the LAST OF /N/ with the word "CENTS" tacked on:

```
←TO CENTS /N/
>1Ø OUTPUT SENTENCE OF LAST OF /N/ AND "CENTS"
>END
←
```

Likewise, LAST OF BUTLAST OF /N/ will provide the tens place and BUTLAST OF BUTLAST OF /N/ will represent the number of hundreds of any modern number. DIMES and DOLLARS follow accordingly.

```
←TO DIMES /N/
>1Ø OUTPUT SENTENCE OF (LAST OF BUTLAST OF /N/) AND "DIMES"
>END
←

←TO DOLLARS /N/
>1Ø OUTPUT SENTENCE OF (BUTLAST OF BUTLAST OF /N/) AND "DOLLARS"
>END
←
```

The conversion from money back to modern numbers is just as straightforward using three procedures, UNDOLLARS, UNDIMES, and UNCENTS which take as input money numbers in standard form and output the modern number representing, respectively, the number of dollars, dimes, and cents in the input. Thus, for example, UNDIMES OF "3 DOLLARS 7 DIMES 4 CENTS" is "7".

The conversion procedure UNMONEY needs only put these three outputs together as a word to get the modern number equivalent of its money number input.

```
←TO UNMONEY /N/
>1Ø OUTPUT WORD OF (UNDOLLARS OF /N/) AND WORD OF (UNDIMES
    OF /N/) AND (UNCENTS OF /N/)
>END
←
```

The three subprocedures, UNDOLLARS, UNDIMES, and UNCENTS, just chop the appropriate number of words off their six-word sentence input to get their outputs.

```
←TO UNCENTS /N/
>1Ø OUTPUT LAST OF BUTLAST OF /N/
>END
←

←TO UNDIMES /N/
>1Ø OUTPUT FIRST OF BUTFIRST OF BUTFIRST OF /N/
>END                                  (or, equivalently, LAST OF BUTLAST
←                                      OF BUTLAST OF BUTLAST OF /N/)

←TO UNDOLLARS /N/
>1Ø OUTPUT FIRST OF /N/               (or, equivalently, LAST OF BUTLAST
>END                                   OF BUTLAST OF BUTLAST OF BUTLAST
←                                      OF BUTLAST OF /N/)
```

We are now ready to consider the problem of adding two money numbers. As with place numbers, we will first write a MONEYADD that adds currency without any carrying from one denomination to another, and later write a procedure that performs the carrying by putting non-standard money numbers into standard form.

When we added place numbers we found that the sum of two place numbers is the same as the sum of the ones plus the sum of the tens plus the sum of the hundreds and so on. Likewise, the sum of two money numbers will be the sum of the cents plus the sum of the dimes plus the sum of the dollars. Now, the sum, for example, of the cents of the two money numbers "6 DOLLARS 4 DIMES 3 CENTS" and "15 DOLLARS 8 DIMES 5 CENTS", is simply the sentence of the sum of the UNCENTS of the numbers "3" and "5" and the word "CENTS". To perform the addition of the two digits provided by UNCENTS, we can use the modern number ADD procedure we wrote in terms of the procedure NEXT. Thus, the sum of the cents of two money numbers, /M/ and /N/, is the SENTENCE OF ADD OF (UNCENTS OF /M/)

AND (UNCENTS OF /N/) AND "CENTS", in our example, SENTENCE OF (ADD OF "3" AND "5") AND "CENTS" or "8 CENTS". The sum of the dollars and dimes is computed in precisely the same way. Thus, our MONEYADD need only stick the three two-word sentences together to get the final answer.

```
←TO MONEYADD /M/ AND /N/
>1Ø OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF SENTENCE OF
    SENTENCE OF ADD OF (UNDOLLARS OF /M/ AND UNDOLLARS OF /N/)
    AND "DOLLARS" AND ADD OF (UNDIMES OF /M/ AND UNDIMES OF
    /N/) AND "DIMES" AND ADD OF (UNCENTS OF /M/ AND UNCENTS OF
    /N/) AND "CENTS"
>END
←
```

With MONEYADD written, we now need a procedure that puts its non-standard outputs into standard form. Since standard form for a money number means no more than nine cents and no more than nine dimes (in order to be able to represent any amount of money, we have to allow any number of dollars), the process of standardization will be essentially that of taking piles of change and returning the same number in the largest denominations possible. Thus we will call our procedure TAKECHANGE.

A money number in standard form, as we just noted, has no more than nine cents or nine dimes. These are the two conditions for which we must test. Now, if there *are* more than nine cents in a number, we want to extract from the cents' place as many groups of ten cents as possible, and add, for each group of ten, one dime to our pile of dimes. Likewise, an excess of dimes is handled by increasing our number of dollars by one for each group of ten dimes, and leaving the leftover in the dimes place. This is just like what we did to non-standard place numbers in STANDARDIZE, except there we had to continue the process for all the places in the number, and here we only have to worry about the dimes and cents.

When we wrote STANDARDIZE, it was necessary to define two sub-procedures, TENS and ONES, to extract from a place the number of groups of ten X's and the leftover X's, respectively. Clearly we will need something like this for TAKECHANGE. But, modern numbers are so designed that numbers smaller than 1Ø (9, 8, 7, etc.) are all one character long and numbers larger than 9 (1Ø, 11, 12, 13 ... 1ØØ, 1Ø1, etc.) are all two or more characters long. Thus, if BUTLAST OF /N/ is /EMPTY/, we know that /N/ is 9 or smaller. Similarly, a non-/EMPTY/ BUTLAST OF /N/ contains all the groups of ten in /N/ (e.g., BUTLAST OF "62" is "6"), and LAST OF /N/ contains the leftover after full groups of ten are extracted (e.g., LAST OF "62" is "2"). Using UNCENTS, UNDIMES, and UNDOLLARS to get the digital parts of a money number, TAKECHANGE goes like this:

```
←TO TAKECHANGE /N/
>1Ø TEST EMPTYP OF BUTLAST OF UNCENTS (Do we have less than 1Ø
    OF /N/                             cents?)
>2Ø IF FALSE MAKE "N" SENTENCE OF     (If not, add the full
    SENTENCE OF SENTENCE OF SENTENCE   groups of ten cents to
    OF SENTENCE OF (UNDOLLARS OF /N/)  the dimes, leaving the
    AND "DOLLARS" AND ADD OF (UNDIMES  remaining nine or fewer
    OF /N/) AND (BUTLAST OF UNCENTS OF pennies in the cents
    /N/) AND "DIMES" AND (LAST OF      place)
    UNCENTS OF /N/) AND "CENTS"
>3Ø TEST EMPTYP OF BUTLAST OF UNDIMES (Do we have less than 1Ø
    OF /N/                             dimes?)
>4Ø IF FALSE MAKE "N" SENTENCE OF     (If not, add the full groups
    SENTENCE OF SENTENCE OF SENTENCE   of ten dimes to the dollars,
    OF SENTENCE OF ADD OF (UNDOLLARS   leaving out the remaining
    OF /N/) AND (BUTLAST OF UNDIMES    dimes in the tens place)
    OF /N/) AND "DOLLARS" AND (LAST
    OF UNDIMES OF /N/) AND "DIMES"
    AND UNCENTS OF /N/ AND "CENTS"
>5Ø OUTPUT /N/                        (The number in standard form)
>END
←
```

Now a procedure that adds two money numbers and gives the sum in standard form can be written using MONEYADD and TAKECHANGE:

```
←TO NEWMONEYADD /M/ AND /N/
>1Ø OUTPUT TAKECHANGE OF MONEYADD OF /M/ AND /N/
>END
←
```

6.5 Mod Number Addition

How does one add two modern numbers, performing carries as they occur? We know from our experience with two-part addition algorithms (STANDARDIZE OF PLACEADD and TAKECHANGE OF MONEYADD) that one wants to add the values of corresponding places, starting with the right-most and working, one place at a time, to the left-most place. We also know from these algorithms that when we come across a place whose contents are greater than nine, we want to add the groups of ten in that place to the next place on the left and leave the remainder where it was. The standard procedure for adding two numbers on paper combines these two processes into a single algorithm.

Let's try to write a mod number add procedure, SUM. SUM should follow fairly directly from the above discussion. SUM OF "362" AND "275" begins by adding (using ADD) the LASTs of the two numbers, "2" and "5". Since their sum, "7", does not exceed "9" (BUTLAST OF "7" is /EMPTY/), the original problem is equivalent to finding the WORD OF SUM OF (BUTLAST OF "362") AND (BUTLAST OF "275") AND "7". (That is, the problem is reduced to the simpler problem: SUM OF "36" AND "27".) SUM OF "36" AND "27", in turn, proceeds by adding the LASTs of the two inputs, or "6" and "7". However, since their sum, "13", *does* exceed "9" (BUTLAST OF "13" is not /EMPTY/), this problem must be rewritten as the WORD OF SUM OF (NEXT OF BUTLAST OF "36") AND (BUTLAST OF "27") AND LAST OF "13",

or, WORD OF (SUM OF "4" AND "2") AND ("3"). SUM OF "4" AND "2" is "6", and, our numbers having been reduced to /EMPTY/, we get the final answer "637". Thus, SUM is defined as follows:

```
←TO SUM /M/ AND /N/
>1Ø TEST EMPTYP OF /M/                          (If either number is
>2Ø IF TRUE OUTPUT /N/                           /EMPTY/, their sum is
>3Ø TEST EMPTYP OF /N/                           simply the other number)
>4Ø IF TRUE OUTPUT /M/
>5Ø MAKE
      NAME:  "LASTNUM"
      THING:  ADD OF (LAST OF /M/) AND
              (LAST OF /N/)
>6Ø TEST EMPTYP OF BUTLAST OF                   (Is this sum less than 9,
   /LASTNUM/                                     i.e., a single digit?)
>7Ø IF TRUE OUTPUT WORD OF (SUM OF              (If so, append this digit
   BUTLAST OF /M/ AND BUTLAST OF /N/)            to the SUM of the rest
   AND /LASTNUM/                                 of the numbers)
>8Ø OUTPUT WORD OF (SUM OF NEXT OF              (If not, append it's last
   BUTLAST OF /M/ AND BUTLAST OF /N/)            digit to the SUM of the
   AND (LAST OF /LASTNUM/)                       rest of the two numbers
                                                 incremented by 1)
>END
←
```

6.6 Mod Number Subtraction

By taking advantage of positional information, faster procedures can be written for adding, subtracting, or comparing numbers by operating upon them one place at a time, then by simply counting up to the answer using ADDX or NEXT. The procedure DIF subtracts two modern numbers by counting up to their difference. It operates quite rapidly for single digits, but with even moderately large numbers, seems very slow. We will improve DIF by writing the mod number equivalent of PLACESUB, called DIFFERENCE. This new subtraction procedure for modern numbers will find the difference of corresponding digits in each successive place of the numbers, from right to left, "borrowing" from the next place to the left when necessary. When all the places have been exhausted, it will output the word of the partial differences.

The procedure DIFFERENCE OF /M/ AND /N/ follows PLACESUB very closely. If it were not for the cases that involve borrowing, we could find the difference (using DIF) of the LASTs of the two words directly, and tack that partial difference onto the end of the DIFFERENCE of the BUTLASTs of the numbers, continuing recursively until /N/ becomes /EMPTY/. Thus, the DIFFERENCE OF "564" AND "321" would be computed as the DIF of "4" and "1" ("3") tacked onto the end of the DIFFERENCE of "56" and "32". The latter problem, in turn, can be rewritten as the WORD of the DIFFERENCE OF "5" and "3" and the DIF OF "6" and "2" ("4"), and the final solution "243" follows trivially. This non-borrowing DIFFERENCE can be written,

```
←TO DIFFERENCE /M/ AND /N/
>1Ø TEST EMPTYP OF /N/                    (If /N/ is /EMPTY/, the
>2Ø IF TRUE OUTPUT /M/                     answer is /M/)
>3Ø OUTPUT WORD OF DIFFERENCE OF (BUTLAST
    OF /M/) AND (BUTLAST OF /N/) AND DIF
    OF (LAST OF /M/) AND (LAST OF /N/)
>END
←
```

In order to make DIFFERENCE work for numbers which contain places in /M/ which are smaller than the corresponding places in /N/, we must examine the LASTs of the two numbers before we operate on them. If LAST OF /M/ is greater than or equal to LAST OF /N/, then we can go ahead with the algorithm as just defined. If, on the other hand, LAST OF /N/ is bigger than LAST OF /M/, then, as in the place number subtraction procedure PLACESUB, we will have to "borrow" a unit from the next place to the left of /M/ (using a ONELESS procedure for mod numbers), make good the loan by adding ten units to the LAST OF /M/, and, after performing the new subtracting, go on to DIFFERENCE the rest of the numbers. Since we know each place will contain exactly one digit, the ten units can

be quickly added to the LAST OF /M/ simply by WORDing "1" to the LAST OF /M/. Thus, the DIFFERENCE procedure with borrowing may be written as follows.

```
←TO DIFFERENCE /M/ AND /N/
>1Ø TEST EMPTYP OF /N/
>2Ø IF TRUE OUTPUT /M/
>3Ø TEST IS COMPARE OF (LAST OF /M/) AND (Is the LAST of /N/ bigger
    (LAST OF /N/) "SECOND"                 than LAST of /M/?)
>4Ø IF FALSE OUTPUT WORD OF DIFFERENCE
    OF (BUTLAST OF /M/) AND (BUTLAST       (If not, subtract without
    OF /N/) AND DIF OF (LAST OF /M/)        borrowing)
    AND (LAST OF /N/)
>5Ø IF TRUE OUTPUT WORD OF DIFFERENCE      (If so, borrow ten from
    OF (ONELESS OF BUTLAST OF /M/)          the rest of the number,
    AND (BUTLAST OF /N/) AND DIF OF         add it to LAST of /M/,
    (WORD OF "1" AND LAST OF /M/)           and then proceed with the
    AND (LAST OF /N/)                       subtraction)
>END
←
```

The modern number ONELESS procedure is very much like its place number equivalent. If the last place in the input /N/ contains anything but "Ø", then ONELESS will output the WORD of BUTLAST OF /N/ and the DIF OF LAST OF /N/ AND "1". When LAST OF /N/ is "Ø", DIF OF LAST OF /N/ AND "1" is undefined, so ONELESS must diminish the BUTLAST OF /N/ by one unit, convert the borrowed unit into the "1Ø" it represents in the LAST place, and then perform the subtraction by "1". Thus, if LAST OF /N/ is "Ø", ONELESS will output the WORD of ONELESS OF BUTLAST OF /N/ and the DIF OF "1Ø" AND "1", or "9". For example, ONELESS OF "43Ø" is the WORD of (ONELESS OF "43") and "9", or WORD OF "4" AND (WORD OF DIF OF "2" AND "1") AND "9", or "429".

```
←TO ONELESS /N/
>1Ø  TEST ZEROP OF LAST OF /N/          (Is the last place a "Ø"?)
>2Ø  IF TRUE OUTPUT WORD OF (ONELESS    (If so, borrow from the
     OF BUTLAST OF /N/) AND "9"          BUTLAST)
>3Ø  OUTPUT WORD OF (BUTLAST OF         (If not, just diminish the
     /N/) AND (DIF OF LAST OF /N/        last place by "1")
     AND "1")
>END
←
```

Before proceeding with multiplication and division of modern numbers, the built-in LOGO procedures COUNT, SUM, and DIFF should be introduced. These built-in procedures for addition and subtraction, SUM and DIFF, will give the same outputs as the corresponding ones we defined, but are written in machine coding and thus are considerably faster. We will need this added speed when performing the repeated addition and subtraction involved in carrying out multiplication and division.

6.7 Multiplication of Mod Numbers

The multiplication procedure we studied for mark numbers substituted for every mark in /N/, the word /M/. Thus the product of /M/ and /N/ was the WORD of /N/ copies of /M/. We can make an analogous procedure for multiplying modern numbers. Since SUM has the same role for modern numbers as WORD has for mark numbers, the product of the modern numbers /M/ and /N/ can be computed by finding the SUM of /N/ copies of /M/. For example, the product of "6" and "3" can be expressed as "6+6+6" or "18". For example,

```
45 * 1 = 45
45 * 2 = 45 + 45 = 9Ø
45 * 3 = 9Ø + 45 = 135
45 * 4 = 135 + 45 = 18Ø
45 * 5 = 18Ø + 45 = 225
45 * 6 = 225 + 45 = 27Ø
```

A procedure, MULTIPLY, for computing the product of two modern numbers, /M/ and /N/, recursively, by successive addition can be easily written if we recognize that the product of /M/ and /N/ is the same as the SUM OF /M/ and the product of /M/ and (/N/-1). For example, MULTIPLY OF "4" AND "2" will be the same as SUM OF "4" AND MULTIPLY OF "4" AND "1". MULTIPLY OF "4" AND "1", in turn, can be rewritten as SUM OF "4" AND MULTIPLY OF "4" AND "Ø". If MULTIPLY outputs "Ø" when /N/ is "Ø", the final result will be the SUM OF "4" AND SUM OF "4" AND "Ø", or "8", the desired answer.

```
←TO MULTIPLY /M/ AND /N/
>1Ø TEST ZEROP OF /N/
>2Ø IF TRUE OUTPUT "Ø"
>3Ø OUTPUT SUM OF /M/ AND MULTIPLY OF /M/ AND
    (DIFF OF /N/ AND "1")
>END
←
```

If you run MULTIPLY for large /N/'s, you will probably find it slower than you wish. It takes a long time to perform, say 325 additions. We can go a long way toward alleviating this problem by taking advantage of a feature built into the positional notation for modern numbers.

Here's an interesting phenomenon: MULTIPLY OF "1Ø" AND "2" outputs "2Ø"; MULTIPLY OF "34" AND "1Ø" outputs "34Ø"; MULTIPLY of "128" AND "1Ø" outputs "128Ø". In fact, the product of "1Ø" and any number /M/, is just the WORD OF /M/ AND "Ø". That is a neat shortcut. Similarly, the product of "1ØØ" and any /M/ is the WORD OF /M/ AND "ØØ". The same is true of multiplication by 1ØØØ, 1ØØØØ, and so on. These shortcuts are not surprising if we remember the meaning of place values in positional notations. When you shift every digit in a number one place to the left by sticking a "Ø" on the end of it, you are, in effect, moving the digit in the ones place into the tens place, the digit in the tens place into the hundreds, and so forth all the way down the number. Thus, for example, a "2" representing two tens becomes a "2" representing two hundreds, clearly making the number ten times bigger. Note that if we had chosen to define our place values as, say, five times the value of the previous place, then sticking a Ø on the end of a number would have had the effect of multiplying the number by "5". Note also that if we had chosen a different place value relationship for each place (i.e, a variable radix instead of a fixed one), such shortcuts would not be possible.

The discovery of an easy way to multiply numbers by powers of 1Ø is an important step towards the development of a faster multiplication algorithm. We can now multiply, say, "864" by "2ØØ", by adding "864" to itself, and appending "ØØ" to the end of the

sum. This is a considerable improvement over SUMming "864" two hundred times.

The second step in making a speedier multiply uses the fact that, for any numbers M, N, O, P; M*(N+O+P) has the same value as (M*N)+(M*O)+(M*P). Using this, we can find the product of, say, "872" and "245" by first rewriting the problem as the product of "872" and ("2ØØ"+"4Ø"+"5") and then transforming this into the sum of the products of "872" and "2ØØ", "872" and "4Ø", and "872" and "5". This is, in fact, the principle behind the familiar long multiplication algorithm which is used to solve the above problem in this form.

```
  872
 X245
 436Ø    (5*872)
3488      (4*872)
1744       (2*872)
213640    (245*872)
```

The indentation of the partial products has the same effect as appending the proper number of Ø's onto the end of each of them.

```
  872
 X245
 436Ø    (5*872)
3488Ø    (4Ø*872)
174400   (2ØØ*872)
213640    (245*872)
```

The demonstration program DEMOFASTMULT (see appendix) uses this algorithm with any two mod number inputs. The printout shows the operation of the algorithm, producing the successive partial products as these are developed.

We can now write a LOGO procedure, MULT, that will utilize these two tricks to obtain the product of two modern numbers quickly.

For the numbers, "872" and "245", we want MULT to multiply "872" by "5" (using MULTIPLY), add that to "872*1Ø*4" (or, 872Ø*4), and add that to "872*1ØØ*2" (or, 872ØØ*2). In general, then, MULT will MULTIPLY /M/ AND LAST OF /N/ and add that product to MULT OF (WORD OF /M/ AND "Ø") AND BUTLAST OF /N/. When /N/ becomes /EMPTY/, MULT should output "Ø".

```
←TO MULT /M/ AND /N/
>1Ø TEST EMPTYP OF /N/
>2Ø IF TRUE OUTPUT "Ø"
>3Ø OUTPUT SUM OF MULTIPLY OF /M/ AND (LAST OF
    /N/) AND MULT OF (WORD OF /M/ AND "Ø")
    AND (BUTLAST OF /N/)
>END
←
```

A TRACE of MULT will help to clarify its operation:

```
←TRACE MULT
←PRINT MULT OF "872" AND "245"
MULT OF "872" AND "245"
  MULT OF "872Ø" AND "24"
    MULT OF "872ØØ" AND "2"
      MULT OF "872ØØØ" AND ""
      MULT OUTPUTS "Ø"
    MULT OUTPUTS "174ØØ" (872ØØ*2)
  MULT OUTPUTS "2Ø928Ø" (872ØØ*2)+(872Ø*4)
MULT OUTPUTS "213640" (872ØØ*2)+(872Ø*4)+(872*5)
213640
←
```

Students will wish to compare the time required to multiply two mod numbers using MULTIPLY with the time using MULT. By trying this with a range of numbers, including some large or even moderately-large ones, one can judge the payoff of the increase in complexity required to achieve the associated increase in speed. One can make an interesting comparison along these lines using relatively small numbers as inputs in a process requiring repeated multiplication.

Thus, consider the following procedure for computing the factorial of a number (N!=N*(N-1)!; Ø!=1):

```
←TO FACTORIAL /N/
>1Ø TEST IS /N/ Ø
>2Ø IF TRUE OUTPUT "1"
>3Ø OUTPUT MULTIPLY OF /N/ AND (FACTORIAL
    OF DIFF OF /N/ AND 1)
>END
←
```

Even with relatively small inputs such as "6", the FACTORIAL procedure takes an appreciable time to finish. But, if we replace MULTIPLY by MULT in line 3Ø of FACTORIAL, the modified FACTORIAL procedure will be sensibly faster. For example, for /N/="6" the new FACTORIAL procedure will be about four times faster. Moreover, the difference goes up rather steeply as /N/ increases.

6.8 Division of Mod Numbers

Considered from one aspect, division means finding how many copies of one number (the "divisor") are contained in a second number (the "dividend"). One example of division is the problem of choosing-up sides for a baseball game. The two team captains represent the divisor and the pool of players to be chosen represents the dividend. When the captains take pairs of players our of the pool until there are no more pairs left, they are, in essence, dividing the group of players by "2". If, after the division process is completed, there remains an odd man, he represents the "remainder" of the division. The number of players on each team represents the "quotient". Dealing out a deck of fifty-two cards to three people is an example of division by three. The number of cards each person receives (seventeen) is the quotient and the one card left over is the remainder.

Division with mark numbers means chopping from the /DIVIDEND/ as many copies of the /DIVISOR/ as possible. The /QUOTIENT/, or number of possible chops, starts off at "Ø" and is increased by "X" each time another copy of /DIVISOR/ is removed. When the /DIVIDEND/ (which gets shorter by /DIVISOR/ each time the process is executed) becomes smaller than the /DIVISOR/, no more chops are possible and the process is completed. /QUOTIENT/ then contains the proper answer and whatever is left in /DIVIDEND/ is the remainder. For example, to divide "XXXXXXX" by "XXX", we chop two ("XX") copies of "XXX" off "XXXXXXX" and are left with a remainder of "X". The LOGO procedure we wrote for division of mark numbers outputs the answer as a sentence of the quotient and the remainder, in this case, "XX X".

Division of modern numbers is entirely analogous to mark number division. Since "chopping" in mark numbers is equivalent to subtracting in any kind of place notation, dividing the modern number /DIVIDEND/ by a mod /DIVISOR/ means repeatedly subtracting /DIVISOR/ from /DIVIDEND/ (while simultaneously increasing /QUOTIENT/ by "1") until /DIVIDEND/ is smaller than /DIVISOR/. The LOGO procedures for division of modern numbers are thus very much like those we wrote for mark numbers. To write DIVIDE, we use a subprocedure called DIV.

```
←TO DIVIDE /DIVIDEND/ AND /DIVISOR/
>1Ø OUTPUT DIV OF /DIVIDEND/ AND /DIVISOR/ AND "Ø"
>END
←
```

DIV is a recursive procedure of three inputs, /DIVIDEND/ (which gets smaller by /DIVISOR/ each time through; /DIVISOR/ (which remains constant throughout); and /QUOTIENT/ (which starts at "Ø", incremented by "1" each time through).

```
←TO DIV /DIVIDEND/ AND /DIVISOR/ AND /QUOTIENT/
>1Ø TEST GREATERP OF /DIVISOR/ AND  (Is the divisor greater than
   /DIVIDEND/                        the quotient?)
>2Ø IF TRUE OUTPUT SENTENCE OF      (If so, we are finished)
   /QUOTIENT/ AND /DIVIDEND/
>3Ø MAKE
      NAME:  "NEWDIVIDEND"          (Otherwise, subtract /DIVISOR/
      THING:  DIFFERENCE OF          from /DIVIDEND/ to get the
              /DIVIDEND/ AND /DIVISOR/   new dividend)
>4Ø MAKE
      NAME:  "NEWQUO"               (And increment the partially
      THING:  SUM OF /QUOTIENT/      developed /QUOTIENT/ by 1)
              AND "1"
>5Ø OUTPUT DIV OF /NEWDIVIDEND/     (And continue the process with
   AND /DIVISOR/ AND /NEWQUO/        the new dividend and quotient)
>END
←
```

A TRACE of DIV shows the progressive changes in /DIVIDEND/ and /QUOTIENT/ throughout the division process.

```
←TRACE DIV
←PRINT DIVIDE OF "23" AND "5"
DIV OF "23" "5" "Ø"
  DIV OF "18" "5" "1"
    DIV OF "13" "5" "2"
      DIV OF "8" "5" "3"
        DIV OF "3" "5" "4"
        DIV OUTPUTS "4 3"
      DIV OUTPUTS "4 3"
    DIV OUTPUTS "4 3"
  DIV OUTPUTS "4 3"
DIV OUTPUTS "4 3"
4 3
←
```

```
/DIVIDEND/ /DIVISOR/=5 /QUOTIENT/
    23                      Ø
  23-5=18                  Ø+1=1
  18-5=13                  1+1=2
  13-5=8                   2+1=3
(/DIVIDEND/ now less than /DIVISOR/
 so DIV outputs the SENTENCE of
 /QUOTIENT/ and /DIVIDEND/)
(The answer is passed all the way
 back to the first DIV which
 inputs it to PRINT)
```

The standard method for checking the result of a division is to multiply the quotient by the divisor and compare that product with the original dividend. If the division was properly executed, the difference between the two should equal the remainder. Trying this with the above example, we multiply "4" by "5" and get "2Ø". Since the remainder was "3" and the original dividend was "23", it appears that our division was correct.

This relationship between mod multiplication and mod division also holds between mark number multiplication and mark number division, but it may be easier to understand with mark numbers. For example, we multiply "XXXX" by "XX" by substituting "XX" for for each mark in "XXXX". The product of "XXXX" and "XX" is, thus, the WORD composed of "XX" and "XX" and "XX" and "XX", or, "XXXXXXXX". We divide "XXXXXXXX" by "XX", in turn, by chopping off from "XXXXXXXX" as many full copies of "XX" as we can, or "XXXX" full copies. Thus multiplication (substitution) has the opposite effect as division (chopping) on mark numbers. In precisely the same sense, MULTIPLY has the opposite effect on mod numbers as DIVIDE. In general, given any two mod numbers /M/ and /N/, if multiplying /M/ and /N/ gives "PRODUCT", then dividing /PRODUCT/ by /M/ will give /N/ (or, alternatively, dividing /PRODUCT/ by /N/ will give /M/).

DEMODIV (see appendix) is a demonstration procedure for mod number division. It permits the user to make a series of guesses of the quotient and multiplies each guess by the divisor, to guide the user towards the answer. For example,

```
←DEMODIV "6524" AND "42"
HOW MANY TIMES DOES 42 GO INTO 6524?  1ØØ
TOO SMALL!  42 TIMES 1ØØ IS ONLY 42ØØ.
HOW MANY TIMES DOES 42 DO INTO 6524?  2ØØ
TOO BIG!  42 TIMES 2ØØ IS 84ØØ.
HOW MANY TIMES DOES 42 GO INTO 6524?  15Ø
TOO SMALL!  42 TIMES 15Ø IS ONLY 63ØØ.
HOW MANY TIMES DOES 42 GO INTO 6524?  16Ø
TOO BIG!  42 TIMES 16Ø IS 672Ø.
HOW MANY TIMES DOES 42 GO INTO 6524?  155
RIGHT!  42 GOES INTO 6524 155 TIMES WITH 14 LEFT OVER.
←
```

Use of DEMODIV gives the user practice in estimating quotients, and at the same time may help him see the need for an efficient method for zooming in on the correct answer.

DEMOLONGDIV (also in the appendix) shows a different approach to division of mod numbers using both multiplication and subtraction. It allows the user to make any guess (except guesses that are too big) at the quotient, subtracts that many copies of the divisor from the dividend (i.e., changes the previous /DIVIDEND/ to DIFF OF /DIVIDEND/ AND MULT OF /DIVISOR/ AND /GUESS/) and repeats the process with the new dividend. When the dividend is less than the divisor, DEMOLONGDIV outputs the final quotient, the sum of all the partial quotients. In addition, at each stage of the process, DEMOLONGDIV prints the user's guess, along with the associated multiplication and subtraction performed by the computer in standard "long-division" format. Thus, for example,

```
        2
12 )--62--
     24
     --
     38
```

is printed after the user makes a guess of "2" for the division of "62" by "12", and shows the subtraction of "2" copies of "12" (or, "24") from the dividend, leaving a new dividend of "38".

Both demonstration procedures are useful. The unsophisticated one, DEMODIV, makes it easy for the user to do division problems, lots of them. It can be valuable in motivating him to find an efficient method for making better guesses. DEMOLONGDIV provides a natural introduction to the standard "long division" procedure.

It is reasonable to ask whether we can make a faster division procedure that uses the trick of subtracting large numbers of copies of the divisor at each stage. In writing the procedure MULT, we noted that multiplying a number by powers of ten is easily accomplished by tacking Ø's onto the end of the number. Thus, if we know approximately how many Ø's to tack onto the end

of the divisor, we can construct a word with that many Ø's, say /ZEROS/. We can then subtract this WORD OF /DIVISOR/ AND /ZEROS/ (which represents the product of /DIVISOR/ and the power of ten corresponding to the number of zeros in /ZEROS/) from /DIVIDEND/, while increasing /QUOTIENT/ by the WORD OF "1" AND /ZEROS/ (that same power of ten). If the /DIVIDEND/ is too small for such a subtraction to be performed, we can just replace /ZEROS/ by BUTLAST OF /ZEROS/ and try again. Thus, the number we subtract will have the largest order of magnitude possible. As in the earlier DIV, when /DIVIDEND/ becomes smaller than /DIVISOR/, the process is terminated.

The algorithm sounds difficult but is easy to understand in a particular case. Let's illustrate it with an example, the division of "296Ø" by "24".

Round 1

/DIVIDEND/ is "296Ø"
/DIVISOR/ is "24"
/QUOTIENT/ is "Ø" (the starting value)
/ZEROS/ is "ØØ" (since that is the largest word of Ø's that can be tacked onto "24" without exceeding "296Ø")

Round 2

/DIVIDEND/ is now "296Ø" - "24ØØ" = "56Ø"
(where "24ØØ" is obtained from WORD OF /DIVISOR/ AND /ZEROS/)
/DIVISOR/ is now "24" (unchanged)
/QUOTIENT/ is now "Ø" + "1ØØ" = "1ØØ"
(the SUM of the previous /QUOTIENT/ and WORD OF "1" AND /ZEROS/)

Round 3

/DIVIDEND/ is now "56Ø" - "24Ø" = "32Ø"
(where "24Ø" is obtained from WORD OF /DIVISOR/ AND /ZEROS/)
/DIVISOR/ is now "24" (unchanged)

/QUOTIENT/ is now "1ØØ" + "1Ø" = "11Ø"
(The SUM of the previous /QUOTIENT/ and WORD OF "1" AND /ZEROS/)

/ZEROS/ is now "Ø"
(Since that is the largest word of Ø's that can be tacked onto "24" without exceeding "32Ø")

Round 4

/DIVIDEND/ is now "32Ø" - "24Ø" = "8Ø"
/DIVISOR/ is now "24"
/QUOTIENT/ is now "11Ø" + "1Ø" = "12Ø"
/ZEROS/ is now /EMPTY/ (Since no Ø's can be tacked onto "24" without exceeding "8Ø")

Round 5

/DIVIDEND/ is now "8Ø" - "24" = "56"
/DIVISOR/ is now "24"
/QUOTIENT/ is now "12Ø" + "1" = "121"
/ZEROS/ is now /EMPTY/

Round 6

/DIVIDEND/ is now "56" - "24" = "32"
/DIVISOR/ is now "24"
/QUOTIENT/ is now "121" + "1" = "122"
/ZEROS/ is now /EMPTY/

Round 7

/DIVIDEND/ is now "32" - "24" = "8"
/DIVISOR/ is now "24"
/QUOTIENT/ is now "122" + "1" = "123"
/ZEROS/ is now /EMPTY/

Round 8

Process terminates because /DIVISOR/ is greater than /DIVIDEND/. The answer is the sentence formed by /QUOTIENT/ and /DIVIDEND/, that is "123" 8", meaning the quotient is "123" and the remainder is "8".

We can now write the new DIV procedure.

```
←TO DIV /DIVIDEND/ /DIVISOR/ /QUOTIENT/ /ZEROS/
>1Ø TEST GREATERP OF /DIVISOR/ AND /DIVIDEND/
>2Ø IF TRUE OUTPUT SENTENCE OF /QUOTIENT/ AND /DIVIDEND/
>3Ø TEST GREATERP OF (WORD OF /DIVISOR/ AND /ZEROS/) AND (/DIVIDEND/)
>4Ø IF TRUE OUTPUT DIV OF /DIVIDEND/ AND /DIVISOR/ AND /QUOTIENT/
    AND (BUTLAST OF /ZEROS/)
>5Ø MAKE "NEWDIVIDEND" DIFFERENCE OF (/DIVIDEND/) AND (WORD OF
    /DIVISOR/ AND /ZEROS/)
>6Ø MAKE "NEWQUO" SUM OF /QUOTIENT/ AND (WORD OF "1" AND /ZEROS/)
>7Ø OUTPUT DIV OF /NEWDIVIDEND/ AND /DIVISOR/ AND /NEWQUO/
    AND /ZEROS/
>END
←
```

DIVIDE will now have to call DIV with an extra input, /ZEROS/, which is a function of the lengths of the dividend and the divisor. Assuming we have a procedure ZEROS that takes /DIVIDEND/ and /DIVISOR/ as inputs and computes the proper number of Ø's needed for DIV, DIVIDE can be written like this.

```
←TO DIVIDE /DIVIDEND/ AND /DIVISOR/
>1Ø OUTPUT DIV OF /DIVIDEND/ AND /DIVISOR/ AND "Ø"
    AND (ZEROS OF /DIVIDEND/ AND /DIVISOR/)
>END
←
```

We have written DIV so that it can handle a /ZEROS/ word with too many Ø's in it. (DIV will always chop off the excess - see line 4Ø.) Thus the procedure ZEROS can safely overestimate the number of Ø's needed and will output the largest number of Ø's that could possibly be tacked onto /DIVIDEND/ to make it as long as /DIVISOR/. For example, assuming the divisor is "1Ø" and the dividend "9999", the most efficient subtraction possible by simply tacking Ø's onto the divisor is to take "1ØØØ" from "9999". (The number "1ØØØØ" is too large a subtrahend and the number "1ØØ" is too small.) The number "1ØØØ" is obtained as (WORD OF "1Ø" AND "ØØ"). Thus, the procedure ZEROS should output the word "ØØ". In the example, the divisor is a power of ten and the

dividend a word of all 9's. This is a worst case since it will require the maximum number (nine) of subtractions of the given subtrahend. In general, ZEROS should output the number of Ø's required to make the divisor as long as (having the same COUNT as) the dividend. Recognizing that ZEROS OF /DIVIDEND/ AND /DIVISOR/ is one "Ø" longer than ZEROS OF (BUTLAST OF /DIVIDEND/) AND /DIVISOR/, and that when the COUNTs of the two inputs are equal, ZEROS should output /EMPTY/, the procedure follows easily:

```
←TO ZEROS /DIVIDEND/ AND /DIVISOR/
>1Ø TEST IS (COUNT OF /DIVIDEND/) (COUNT OF /DIVISOR/)
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF "Ø" AND ZEROS OF (BUTLAST OF
    /DIVIDEND/) AND /DIVISOR/
>END
←
```

Let's follow a TRACE of DIV through the example that we discussed above.

```
←TRACE DIV
←PRINT DIVIDE OF "296Ø" AND "24"
DIV OF "296Ø" "24" "Ø" "ØØ"
  DIV OF "56Ø" "24" "1ØØ" "ØØ"    (/DIVIDEND/ has been decreased by
                                    24ØØ, /QUOTIENT/ increased by
                                    "1ØØ")
    DIV OF "56Ø" "24" "1ØØ" "Ø"   (/ZEROS/ has had a Ø dropped off)
      DIV OF "32Ø" "24" "11Ø" "Ø" (/DIVIDEND/ decreased by "24Ø";
                                    /QUOTIENT/ increased by 1Ø)
        DIV OF "8Ø" "24" "12Ø" "Ø"  (/DIVIDEND decreased by "24Ø";
                                     /QUOTIENT/ increased by 1Ø)
DIV OF "8Ø" "24" "12Ø" ""          (/ZEROS/ has been BUTLASTed to
                                    /EMPTY/)
  DIV OF "56" "24" "121" ""
    DIV OF "32" "24" "122" ""
      DIV OF "8" "24" "123" ""     (/DIVIDEND/ smaller than /DIVISOR/
      DIV OUTPUTS "123" 8"          so DIV outputs the answer)
    DIV OUTPUTS "123 8"           (All the way back through the
  DIV OUTPUTS "123 8"              intermediate calls to the first
DIV OUTPUTS "123 8"                call, i.e., for the original
        DIV OUTPUTS "123 8"        problem)
      DIV OUTPUTS "123 8"
```

```
    DIV OUTPUTS "123 8"
  DIV OUTPUTS "123 8"
DIV OUTPUTS "123 8"
123 8                              (PRINT prints the answer)
←
```

As with MULT and MULTIPLY, we can judge the payoff in speed of our new and more complex division procedure over the old, simple DIVIDE by timing their operation with some typical problems. For example, with the division of "9999" by "1ØØ" the old DIVIDE is about 7 times slower than the new one; for "9999" divided by "1Ø" it is more than 40 times slower. The advantage of the new DIVIDE over the old increases even more sharply as the magnitude of the quotients gets still larger.

The built-in LOGO procedures for multiplication and division of mod numbers can be introduced at this point. These are, of course, several orders of magnitude faster than even the most efficient programmed procedures. This greater speed is necessary to make practical the programming of procedures for operations with decimal and fractional numbers.

APPENDICES

EXPLODE: Conversion from Place Numbers to Mark Numbers

Examples

```
←PRINT EXPLODE OF "Ø"

←PRINT EXPLODE OF "XXXXXX"
XXXXXX
←PRINT EXPLODE OF "X XXX"
XXXXXXXXXXXX
←PRINT EXPLODE OF "X Ø Ø"
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
←PRINT EXPLODE OF "X X X X"
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEMOFASTMULT: Fast Multiplication of Modern Numbers

Example

```
←DEMOFASTMULT "872" AND "275"
 872
*275
 ---

IS EQUAL TO 872 * ( 5 + 70 + 200 )
OR
( 872 * 5 ) +
( 872 * 70 ) +
( 872 * 200 )

 872
  *5
 ---
4360

                          ( 872 * 5 ) = 4360

 872
 *70
 ---

EQUALS WORD OF ( 872 * 7 ) AND 0

 872
  *7
 ---
6104

                         ( 872 * 70 ) = 61040

 872
*200
 ---

EQUALS WORD OF ( 872 * 2 ) AND 00

 872
  *2
 ---
1744

                        ( 872 * 200 ) = 174400

   872
  *275
   ---
  4360
 61040
174400
------
239800

                        872 * 275 = 239800
←
```

Listing

```
TO FACTOR /N/ AND /MASK/
10 TEST IS COUNT OF /N/ COUNT OF /MASK/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT SENTENCE OF FACTOR OF /N/ AND WORD OF /MASK/ AND "X" AND
   ZEROS OF /N/ AND /MASK/
END


TO ZEROS /N/ AND /MASK/
10 TEST EMPTYP /MASK/
20 IF TRUE OUTPUT LAST OF /N/
30 OUTPUT WORD OF ZEROS OF BUTLAST OF /N/ AND BUTLAST OF /MASK/ AND "0"

END


TO DASH /N/
10 TEST EMPTYP /N/
20 IF TRUE STOP
30 TYPE "-"
40 DASH BUTLAST OF /N/
END


TO DISTRIBUTE /N/ /FACTORS/
10 TEST EMPTYP BUTFIRST OF /FACTORS/
20 IF TRUE PRINT SENTENCE "(" SENTENCE /N/ SENTENCE "*" SENTENCE
   LAST OF /FACTORS/ ")"
30 IF TRUE STOP
40 PRINT SENTENCE "(" SENTENCE /N/ SENTENCE "*" SENTENCE LAST OF
   /FACTORS/ ") +"
50 DISTRIBUTE /N/ BUTLAST OF /FACTORS/
END


TO DRAW /N/ AND /M/
10 PRINT WORD /BLANK/ /N/
20 BLANK UNWORD OF /N/ AND /M/
30 PRINT WORD OF "*" AND /M/
40 TYPE /BLANK/
50 DASH /N/
60 PRINT ""
END


TO LISTFACT /FACTORS/
10 TEST EMPTYP BUTLAST OF /FACTORS/
20 IF TRUE OUTPUT /FACTORS/
30 OUTPUT SENTENCE SENTENCE OF LAST OF /FACTORS/ AND "+" AND LISTFACT
   OF BUTLAST OF /FACTORS/
END
```

```
TO DEMOMULT2 /N/ /FACTORS/ /ANS/
10 TEST EMPTYP /FACTORS/
20 IF TRUE OUTPUT /ANS/
30 PRINT WORD OF /SKIP/ AND /SKIP/
40 MAKE "MM" LAST OF /FACTORS/
50 TEST ZEROP LAST OF /MM/
60 IF FALSE GO TO LINE "110"
70 DRAW /N/ /MM/
80 PRINT ""
90 PRINT SENTENCE OF "EQUALS WORD OF (" SENTENCE OF /N/ SENTENCE OF "*"
   SENTENCE OF FIRST OF /MM/ SENTENCE OF ") AND" BUTFIRST OF /MM/
100 PRINT ""
110 DRAW /N/ AND FIRST OF /MM/
120 MAKE "TEMPROD" MULT OF /N/ AND FIRST OF /MM/
140 PRINT WORD OF /TEMPROD/ AND /SKIP/
150 MAKE "ANS" SENTENCE OF /ANS/ AND WORD OF /TEMPROD/ AND BUTFIRST
    OF /MM/
160 PRINT SENTENCE OF WORD OF /TAB/ AND "(" SENTENCE /N/ SENTENCE "*"
    SENTENCE /MM/ SENTENCE ") =" LAST OF /ANS/
170 PRINT ""
180 MAKE "FACTORS" BUTLAST OF /FACTORS/
190 GO TO LINE "10"
END


TO BLANK /N/
10 TEST EMPTYP /N/
20 IF TRUE STOP
30 TYPE /BLANK/
40 BLANK BUTLAST OF /N/
END


TO UNWORD /N/ AND /M/
10 TEST EMPTYP /M/
20 IF TRUE OUTPUT /N/
30 OUTPUT UNWORD BUTLAST OF /N/ BUTLAST OF /M/
END


TO LINEUP /ANS/
10 TEST EMPTYP /ANS/
20 IF TRUE STOP
30 BLANK UNWORD OF LAST OF /ANS/ AND FIRST OF /ANS/
40 PRINT FIRST OF /ANS/
50 LINEUP BUTFIRST OF /ANS/
END
```

```
TO ADDUP /ANS/
10 TEST EMPTYP /ANS/
20 IF TRUE OUTPUT "0"
30 OUTPUT SUM OF FIRST OF /ANS/ AND ADDUP OF BUTFIRST OF /ANS/
END


TO MULT /N/ AND /M/
10 TEST ZEROP /M/
20 IF TRUE OUTPUT "0"
30 OUTPUT SUM OF /N/ AND MULT OF /N/ AND DIFFERENCE OF /M/ AND "1"
END


TO DEMOFASTMULT /N/ AND /M/
5 PRINT ""
10 DRAW /N/ AND /M/
15 PRINT ""
20 MAKE "FACTORS" FACTOR OF /M/ AND ""
30 PRINT SENTENCE OF "IS EQUAL TO" SENTENCE /N/ SENTENCE "* ("
   SENTENCE LISTFACT OF /FACTORS/ ")"
40 PRINT "OR"
50 DISTRIBUTE /N/ /FACTORS/
60 MAKE "ANS" DEMOMULT2 OF /N/ AND /FACTORS/ AND ""
65 PRINT WORD /SKIP/ /SKIP/
70 BLANK UNWORD OF LAST OF /ANS/ AND /N/
80 PRINT /N/
90 BLANK UNWORD OF LAST OF /ANS/ AND WORD OF "*" AND /M/
100 PRINT WORD OF "*" AND /M/
105 BLANK UNWORD OF LAST OF /ANS/ AND /N/
110 DASH /N/
115 PRINT ""
120 LINEUP /ANS/
140 MAKE "ANS" ADDUP OF /ANS/
150 DASH /ANS/
160 PRINT ""
170 PRINT /ANS/
180 PRINT ""
190 PRINT SENTENCE OF WORD OF /TAB/ AND /N/ SENTENCE "*" SENTENCE /M/
    SENTENCE "=" /ANS/
END
```

DEMODIV: Guessing Quotient of Two Mod Numbers

Examples

```
←DEMODIV "375" AND "4"
HOW MANY TIMES DOES 4 GO INTO 375? 50
TOO SMALL! 4 TIMES 50 IS ONLY 200
HOW MANY TIMES DOES 4 GO INTO 375? 60
TOO SMALL! 4 TIMES 60 IS ONLY 240
HOW MANY TIMES DOES 4 GO INTO 375? 100
TOO BIG! 4 TIMES 100 IS 400
HOW MANY TIMES DOES 4 GO INTO 375? 80
TOO SMALL! 4 TIMES 80 IS ONLY 320
HOW MANY TIMES DOES 4 GO INTO 375? 85
TOO SMALL! 4 TIMES 85 IS ONLY 340
HOW MANY TIMES DOES 4 GO INTO 375? 90
TOO SMALL! 4 TIMES 90 IS ONLY 360
HOW MANY TIMES DOES 4 GO INTO 375? 93
RIGHT! 4 GOES INTO 375 93 TIMES WITH 3 LEFT OVER.
←
```

```
←DEMODIV "6524" AND "42"
HOW MANY TIMES DOES 42 GO INTO 6524? 100
TOO SMALL! 42 TIMES 100 IS ONLY 4200
HOW MANY TIMES DOES 42 GO INTO 6524? 200
TOO BIG! 42 TIMES 200 IS 8400
HOW MANY TIMES DOES 42 GO INTO 6524? 150
TOO SMALL! 42 TIMES 150 IS ONLY 6300
HOW MANY TIMES DOES 42 GO INTO 6524? 160
TOO BIG! 42 TIMES 160 IS 6720
HOW MANY TIMES DOES 42 GO INTO 6524? 155
RIGHT! 42 GOES INTO 6524 155 TIMES WITH 14 LEFT OVER.
←
```

Listing

```
TO DEMODIV /DIVIDEND/ AND /DIVISOR/
10 TYPE SENTENCE OF "HOW MANY TIMES DOES" AND SENTENCE OF /DIVISOR/
   AND SENTENCE OF "GO INTO" AND WORD OF /DIVIDEND/ AND WORD OF "?"
   AND /BLANK/
30 MAKE "QUOTIENT" REQUEST
40 PRINT ""
50 MAKE "PRODUCT" MULT OF /QUOTIENT/ AND /DIVISOR/
60 MAKE "REM" DIFFERENCE OF /DIVIDEND/ AND /PRODUCT/
70 TEST GREATERP /REM/ AND "-1"
80 IF FALSE GO TO LINE "140"
90 TEST GREATERP /REM/ AND /DIVISOR/
100 IF TRUE PRINT SENTENCE OF "TOO SMALL!" AND SENTENCE OF /DIVISOR/
    AND SENTENCE OF "TIMES" AND SENTENCE OF /QUOTIENT/ AND SENTENCE
    OF "IS ONLY" AND /PRODUCT/
110 IF TRUE GO TO LINE "10"
120 PRINT SENTENCE OF "RIGHT!" AND SENTENCE OF /DIVISOR/ AND
    SENTENCE OF "GOES INTO" AND SENTENCE OF /DIVIDEND/ AND SENTENCE
    OF /QUOTIENT/ AND SENTENCE OF "TIMES WITH" AND SENTENCE OF /REM/
    AND "LEFT OVER."
130 STOP
140 PRINT SENTENCE OF "TOO BIG!" AND SENTENCE OF /DIVISOR/ AND
    SENTENCE OF "TIMES" AND SENTENCE OF /QUOTIENT/ AND SENTENCE OF "IS"
    AND /PRODUCT/
150 GO TO LINE "10"
END


TO MULT /N/ AND /M/
10 TEST IS /M/ /EMPTY/
20 IF TRUE OUTPUT "0"
30 OUTPUT SUM OF MULTIPLY OF /N/ AND LAST OF /M/ AND MULT OF WORD OF /N/
    AND "0" AND BUTLAST OF /M/
END


TO MULTIPLY /N/ AND /M/
10 TEST IS /M/ "0"
20 IF TRUE OUTPUT "0"
30 OUTPUT SUM OF /N/ AND MULTIPLY OF /N/ AND DIFFERENCE OF /M/ AND "1"
END
```

DEMOLONGDIV: "Long" Division of Mod Numbers

Examples

```
←DEMOLONGDIV "47354" AND "32"

HOW MANY TIMES DOES 32 GO INTO 47354? 1000
OK. 32 GOES INTO 47354 1000 TIMES WITH 15354 LEFT OVER.

      1000
   ---------
32 ) 47354
     32000
     -----
     15354

HOW MANY TIMES DOES 32 GO INTO 15354? 500
TOO BIG! 32 TIMES 500 IS 16000

HOW MANY TIMES DOES 32 GO INTO 15354? 400
OK. 32 GOES INTO 15354 400 TIMES WITH 2554 LEFT OVER.

       400
   ---------
32 ) 15354
     12800
     -----
      2554

HOW MANY TIMES DOES 32 GO INTO 2554? 80
TOO BIG! 32 TIMES 80 IS 2560

HOW MANY TIMES DOES 32 GO INTO 2554? 70
OK. 32 GOES INTO 2554 70 TIMES WITH 314 LEFT OVER.

       70
   --------
32 ) 2554
     2240
     ----
      314

HOW MANY TIMES DOES 32 GO INTO 314? 9
OK. 32 GOES INTO 314 9 TIMES WITH 26 LEFT OVER.

       9
   -------
32 ) 314
     288
     ---
      26


DONE. 32 GOES INTO 47354 1479 TIMES WITH 26 LEFT OVER.
←
```

Listing

```
TO DRAW /DIVIDEND/ /DIVISOR/ /QUOTIENT/ /PRODUCT/ /REM/
10 MAKE "INDENT" WORD OF /DIVISOR/ AND "X"
20 BLANK WORD OF /INDENT/ AND WORD OF "XX" AND UNWORD OF /DIVIDEND/
   AND /QUOTIENT/
30 PRINT /QUOTIENT/
40 BLANK /INDENT/
50 DASH WORD OF /DIVIDEND/ AND "XXXX"
60 PRINT ""
70 PRINT SENTENCE OF SENTENCE OF /DIVISOR/ AND ")" AND /DIVIDEND/
35 BLANK WORD OF /INDENT/ AND WORD OF "XX" AND UNWORD OF /DIVIDEND/
   AND /PRODUCT/
90 PRINT /PRODUCT/
100 BLANK WORD OF /INDENT/ AND "XX"
110 DASH /DIVIDEND/
115 PRINT ""
120 BLANK WORD OF /INDENT/ AND WORD OF "XX" AND UNWORD OF /DIVIDEND/
    AND /REM/
130 PRINT /REM/
140 PRINT WORD OF /SKIP/ AND /SKIP/
END


TO BLANK /N/
10 TEST EMPTYP /N/
20 IF TRUE STOP
30 TYPE /BLANK/
40 BLANK BUTLAST OF /N/
END


TO UNWORD /N/ AND /M/
10 TEST EMPTYP /M/
20 IF TRUE OUTPUT /N/
30 OUTPUT UNWORD OF BUTLAST OF /N/ AND BUTLAST OF /M/
END


TO DASH /N/
10 TEST EMPTYP /N/
20 IF TRUE STOP
30 TYPE "-"
40 DASH BUTLAST OF /N/
END


TO DEMOLONGDIV /DIVIDEND/ AND /DIVISOR/
10 PRINT SENTENCE "DONE." SENTENCE /DIVISOR/ SENTENCE "GOES INTO"
   SENTENCE /DIVIDEND/ SENTENCE DEMOLONGDIV2 OF /DIVIDEND/ AND /DIVISOR/
    SENTENCE "TIMES WITH" SENTENCE /REM/ "LEFT OVER."
END
```

```
TO MULT /N/ AND /M/
10 TEST EMPTYP /M/
20 IF TRUE OUTPUT "0"
30 OUTPUT SUM MULTIPLY OF /N/ AND LAST OF /M/ AND WORD OF MULT OF /N/
   AND BUTLAST OF /M/ AND "0"
END


TO MULTIPLY /N/ AND /M/
10 TEST ZEROP /M/
20 IF TRUE OUTPUT "0"
30 OUTPUT SUM /N/ MULTIPLY /N/ DIFFERENCE /M/ "1"
END


TO DEMOLONGDIV2 /DIVIDEND/ AND /DIVISOR/
10 TYPE SENTENCE "HOW MANY TIMES DOES" SENTENCE /DIVISOR/ SENTENCE "GO
   INTO" WORD /DIVIDEND/ WORD "?" /BLANK/
20 MAKE "QUOTIENT" REQUEST
30 PRINT ""
40 MAKE "PRODUCT" MULT OF /QUOTIENT/ AND /DIVISOR/
50 MAKE "REM" DIFFERENCE OF /DIVIDEND/ AND /PRODUCT/
60 TEST GREATERP /REM/ "-1"
70 IF FALSE GO TO LINE "130"
80 PRINT SENTENCE "OK," SENTENCE /DIVISOR/ SENTENCE "GOES INTO"
   SENTENCE /DIVIDEND/ SENTENCE /QUOTIENT/ SENTENCE "TIMES WITH"
   SENTENCE /REM/ SENTENCE "LEFT OVER." /SKIP/
90 DRAW /DIVIDEND/ /DIVISOR/ /QUOTIENT/ /PRODUCT/ /REM/
100 TEST GREATERP /REM/ /DIVISOR/
110 IF FALSE OUTPUT /QUOTIENT/ (DONE)
120 OUTPUT SUM OF /QUOTIENT/ AND DEMOLONGDIV2 OF /REM/ AND /DIVISOR/
130 PRINT SENTENCE "TOO BIG!" SENTENCE /DIVISOR/ SENTENCE "TIMES"
    SENTENCE /QUOTIENT/ SENTENCE "IS" SENTENCE /PRODUCT/ /SKIP/
140 GO TO LINE "10"
END
```

LOGO Number Unit

Problems

1. Introduction

This is a collection of student problems and projects for use with the LOGO unit on numbers. The problems follow the teaching text rather closely. They should serve as a starting set just sufficient for working out the main ideas treated in the unit. Teachers will find it easy to include problems of their own, since the material leads quite naturally to excursions of many kinds.

We considered including problems from a variety of additional areas such as figurate numbers, mathematical induction, guessing strategies, and algorithms for approximate arithmetic. It is easy to do so and we plan later to considerably expand the present collection.

The demonstrations referred to in connection with some of the problems are the ones listed in the appendices to the teacher's text. These programs will presumably be made available to the students as a LOGO file. Students will, of course, write their own programs in the course of working on the problems. They will frequently use these programs in turn in solving new problems. To facilitate this mode of use, each student's programs will be stored in computer memory in his personal LOGO file. Thus, starting with this little problem book, students will probably generate a lab workbook of rather impressive size.

2. Mark Numbers

2.1 Introduction to Mark Numbers

(1) Write a procedure that takes a mark number as its input and counts up from that number indefinitely.

(2) Write a similar procedure for counting down by X from a given mark number stopping at /EMPTY/.

2.2 Addition of Mark Numbers

(1) Write a procedure that takes two mark numbers and outputs their sum.

2.3 Comparison of Mark Numbers

(1) Write a procedure for comparing two mark numbers. If the two inputs are the same, it should output "SAME"; otherwise, "FIRST" if the first is larger, "SECOND" if the second is larger.

(2) Write a procedure which takes any two English words and outputs "FIRST", "SECOND", or "SAME" according to which of the words is larger.

(3) Modify this English word comparison procedure to work with French words.

2.4 Subtraction of Mark Numbers

(1) Modify the mark number comparison procedure to output the difference of the two mark numbers.

3. Problems on Compact Numbers

3.1 Grouping Mark Numbers

(1) Write a procedure that counts up from /EMPTY/ in groups of XXXXX, ringing the teletype bell each time it completes a group.

3.2 Conversion Between Mark and Grouped Mark Numbers

(1) Write a procedure that takes a mark number as input and outputs the equivalent grouped mark number.

(2) Write the opposite procedure, which takes a grouped mark number as input and outputs the equivalent mark number.

(3) Using the existing procedure for adding mark numbers, write an addition procedure which takes two grouped mark numbers and outputs as their sum a grouped mark number.

3.3 A Compact Representation of Grouped Numbers

(1) Rewrite the procedure for converting mark numbers to grouped mark numbers so that it now outputs compact numbers.

(2) Extend the new procedure to include compact numbers with H's (an H is equal to ten T's).

(3) Write a procedure that takes a compact number (in X's, T's, and H's) and counts up from that number indefinitely.

3.4 Conversion of Compact Numbers to Mark Numbers

(1) Write a procedure which takes a compact number (in X's and T's) and gives the equivalent mark number.

(2) Extend this procedure to include inputs with H's.

3.5 Disordered Compact Numbers

(1) Modify the compact number to mark number conversion procedure to work with any compact numbers (i.e., not necessarily in standard form).

(2) Write three procedures for finding the number of X's, T's, and H's in any given compact number.

(3) How many different five mark words can you make out of X's and T's?

(4) Write a procedure which takes any compact number (in X's, T's, and H's) and prints out all equivalent mark or compact number representations.

3.6 Standard Form of Compact Numbers

(1) Write a procedure which takes any compact number and outputs the equivalent number in standard form.

(2) Write a procedure that takes any two compact numbers and and produces their sum in standard compact number form.

3.7 Comparing Compact Numbers

(1) Write a procedure that compares any two compact numbers in standard form and outputs "FIRST", "SECOND", or "SAME" according to which is larger.

(2) Extend the procedure to work for any two compact numbers.

3.8 Subtraction of Compact Numbers

(1) Write a procedure which takes two compact numbers in standard form and finds the difference between the number of H's in the two numbers. How closely does this approximate the difference between the two numbers?

(2) Write a procedure that outputs in standard compact number form the difference between two compact numbers in standard form.

4. More About Mark Numbers

4.1 Multiplication of Mark Numbers

(1) Write a procedure that takes a mark number and outputs the product of that number and ten, by substituting "XXXXXXXXXX" for each "X" in the number.

(2) Extend this procedure to multiply any two mark numbers. Call this procedure TIMES.

(3) When will TIMES OF /M/ AND /N/ give the same output as TIMES OF /N/ AND /M/? Which will be faster?

(4) TIMES OF "XX" AND "XXX" gives the same answer as TIMES OF "XXXXXX" AND "X". How many ways can you get TIMES to output "XXXXXXXXXXXXXXXXXXXXXXXX".

4.2 Counting, Copying, and Compact Multiplication

(1) Using TIMES, write a procedure whose output is the number of marks or letters in its input.

(2) Write a two-input procedure that uses TIMES to make the number of copies of the second input that is specified by the first input.

(3) Write a procedure to multiply a mark number by a compact number.

4.3 Division of Mark Numbers

(1) Write a procedure, HALF, to split any mark number into two equal parts.

(2) Write a procedure EVENP whose output is "TRUE" if its input is an even mark number; otherwise its output is "FALSE".

(3) Extend EVENP to work for compact numbers.

(4) Write a procedure for dividing one mark number by another, which outputs the quotient but ignores the remainder.

(5) Write the related division procedure, which outputs the remainder but ignores the quotient.

5. Place Numbers

5.1 Positional Notation

(1) Write a procedure to convert any compact number to an equivalent place number.

(2) Write a procedure to convert any three-position (or smaller) place number to an equivalent compact number.

(3) What is the largest place number that can be printed on a single teletype line. (Assume that one can print one hundred characters on a teletype line.)

(4) Write in place number notation the number of teletype lines that would be required to print out the large place number of problem (1) as a mark number.

(5) Use the demonstration procedure EXPLODE to convert place numbers to equivalent mark numbers.

5.2 Addition of Place Numbers

(1) Write a procedure for adding any two place numbers. Don't worry about carrying.

5.3 The Place Number Successor

(1) Write a procedure that outputs the place number one less than its input.

(2) Using this, write a place number count down procedure which starts from a given input number and terminates with Ø.

5.4 Converting Place Numbers to Standard Form

(1) The addition procedure you wrote can generate place numbers that are not in standard form. Write a procedure that decides whether or not a place number is in standard form.

(2) Write a procedure that takes any place number as an input and outputs the equivalent place number in standard form.

(3) Using this, and your old addition procedure, write an addition procedure that outputs in standard positional notation, the sum of any two place numbers.

5.5 Comparing Place Numbers

(1) Write a procedure to compare two place numbers by converting them to mark numbers, using EXPLODE, and comparing these.

(2) Write a procedure to compare two place numbers (having three positions or less) by converting them to compact numbers and comparing these.

(3) Write a procedure to compare two place numbers, not to find out which one is larger, but simply to find which one is "longer" (i.e., has more places or words, where two numbers with the same number of places are regarded as having the same length).

(4) Write a procedure to compare two place numbers of precisely the same length (in the sense above).

(5) Write a procedure to compare any two place numbers directly.

5.6 Subtraction of Place Numbers

(1) Modify your mark number compare procedure using either your place number successor or predecessor so that it can be used for comparing any two place numbers.

(2) Write a place number addition procedure using your place number successor and predecessor.

(3) Now write a place number subtraction procedure using your place number successor and predecessor.

(4) Write a place number subtraction procedure not using your successor and predecessor that incorporates borrowing.

(5) Compare the speeds of the various place number compare procedures you have written, for both large and small numbers.

(6) Do the same for your several addition and subtraction procedures.

6. Modern Numbers

6.1 Digits

(1) How many one-digit modern numbers are there? How many three-digit modern numbers? How many n-digit modern numbers?

6.2 Conversion Between Mod Numbers and Place Numbers

(1) Write a procedure that takes a mark number less than "XXXXXXXXXX" and outputs the equivalent digit.

(2) Write a procedure that takes a digit and outputs the equivalent mark number.

(3) Write a procedure that takes a place number and outputs the equivalent mod number.

(4) Write the opposite procedure for converting a mod number to an equivalent place number.

6.3 Mod Arithmetic Along the Lines of Place Arithmetic

(1) Using the conversion procedures just written, write mod number successor, predecessor, comparison, addition, and subtraction procedures.

(2) Write a mod number successor procedure directly.

(3) Using this successor, write a mod number count-up procedure that lists successive mod numbers from a given number on.

(4) Similarly, using successor, write procedures for addition, comparison, and subtraction of mod numbers.

(5) Write a procedure that takes a mark number as an input and outputs its mod number equivalent, that is the number of marks in the mark number.

6.4 Adding Money Numbers

(1) Write a procedure to convert any mod number to an equivalent money number.

(2) Write the opposite procedure, converting any money number to the equivalent mod number.

(3) Write a procedure to add two money numbers by converting those to mod numbers and using the existing mod number add procedure without carry.

(4) Write a procedure to convert a money number to standard form, along the lines of the place number standardization procedure.

(5) Combine the add and standardize procedures to write an addition procedure that outputs the sum of two money numbers in standard form.

6.6 Mod Number Subtraction

(1) Write a mod number subtraction procedure that performs borrowing when necessary.

(2) Write a mod number countdown procedure that counts down from a specified number by 1 until Ø.

(3) Write a countdown procedure that counts down from a given number in specified steps (for example, COUNTDOWN 10 3 should print out 1Ø, 7, 4, 1).

6.7 Multiplication of Mod Numbers

(1) Write a mod number multiplication procedure that works by repeated addition.

(2) Write a fast procedure to multiply any mod number by 1ØØØ.

(3) Use the demonstration procedure DEMOFASTMULT.

(4) Write a fast mod number multiplication procedure.

(5) Write a procedure that compares the operating speeds of your slow and fast multiply procedures, generating successive inputs over an extensive range and listing the corresponding multiply times.

(6) Does your fast multiply take the same time if you switch the order of your inputs? Make it do so.

6.8 Division of Mod Numbers

(1) Write a mod number quotient procedure, along the lines of the mark number quotient procedure, ignoring the remainder.

(2) Do the same for a mod number remainder procedure ignoring the quotient.

(3) Combine the two procedures to get a division procedure which outputs a two-word sentence containing the quotient and the remainder.

(4) Use the demonstration procedure DEMODIV. Try to improve your guessing technique.

(5) Use the demonstration procedure DEMOLONGDIV. What are some reasonable guessing strategies?

(6) Write a fast mod number division procedure.

(7) Write a procedure that compares the operating speeds of your slow and fast divide procedures, as you did for multiplication.

Volume 2, Part 2

LOGO FUNCTIONS AND EQUATIONS

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

Wallace Feurzeig

George Lukas

Richard Grant

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

Page

1. Introduction 1

2. Ordered Pairs 2

   2.1 Relations 5

   2.2 Graphing 9

   2.3 Functions 19

   2.4 Coding and Decoding 24

3. Functions Specified by Rules 31

   3.1 Functions as Procedures 31

   3.2 Black Boxes 37

   3.3 Automatic Composition and Generation of Functions . 45

   3.4 Inverse Functions 52

4. Equations 58

   4.1 Evaluating and Testing Expressions 59

   4.2 Generating Equations and Testing Solutions 64

   4.3 Solving Equations 67

   4.4 Inverting Functions 70

   4.5 Harder Equations 76

Problems

Functions and Equations

1. Introduction

The first sections of this unit introduce the notions of relation and function. Functions are specified in two different ways -- by direct enumeration and by rules. Initially a function is regarded as a list of pairs of elements, where each domain element is followed by the corresponding range element. Later a function is represented as a set of rules for determining the output corresponding to any given input. In the first case, functions are expressed as LOGO *lists*, i.e., sentences; in the second case as LOGO *procedures*.

Using the list representation, we develop LOGO procedures for graphing, evaluation, composition, and inversion of functions. Certain interesting functions are not easily expressed as lists, consequently, we develop LOGO procedures as a comprehensive framework for representing functions. Using this representation, the characterization of some functions is conceptually more difficult than with lists. We study this problem directly and, as well, in the context of finding the inverse of a given function, particularly in the case where the function is a composition of functions each of whose inverses are known.

In the last sections of the unit we study the solution of simple algebraic equations. We develop programs for evaluating equations, checking solutions and guessing and searching for solutions. We then propose a solution algorithm based on "undoing" the operation sequence used in generating the equation. By transforming problems of solving equations to problems of inverting functions, we demonstrate the validity of the undoing algorithm for inverting composite functions and solving simple equations.

This material is designed for use at intermediate levels. The unit was written by Wallace Feurzeig and edited by George Lukas. George Lukas and Richard Grant made many valuable contributions throughout.

2. Ordered Pairs

There are at least two distinct ways of expressing a relation between pairs of numbers. One way is by explicit enumeration, such as in a list or table. For example, the table of ordered pairs,

First Number	Second Number
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

is a literal way of giving the binary number representation corresponding to the octal integers.

The elements in a table need not be numbers at all. Thus, the following table

First	Second
A	B
B	C
C	D
D	E
⋮	⋮
Y	Z
Z	A

associates each letter in the English alphabet with its successor (where A is taken as the successor of Z).

One alternate way of writing lists of ordered pairs, is as a LOGO sentence. Thus, the two tables above can be written as follows.

" 0 0 1 1 2 10 3 11 4 100 5 101 6 110 7 111"

"A B B C C D D E ... Y Z Z A"

where rows of corresponding pairs are strung together in a single list. We shall use this notation in the following.

Whether the elements in a table or list of ordered pairs are numbers, letters, or other symbol strings, the use of the list is the same. One "looks up" the (second) element corresponding to a specified (first) element by searching through all the first elements in the list to see if the given element is present. If it is, the element following it is the answer. If not, the pair is not in the list, so we can output the *empty* element as the answer.

The following LOGO procedure, MATCH, carries out this search.

```
TO MATCH /ELEMENT/ /PAIR LIST/
1 TEST EMPTYP /PAIR LIST/          (If the list is exhausted)
2 IF TRUE OUTPUT /EMPTY/           (There is no match)
3 TEST IS /ELEMENT/ FIRST OF       (If the element is the first
  /PAIR LIST/                      element of the first pair on
                                   the list)
4 IF TRUE OUTPUT SECOND OF         (The answer is the second element
  /PAIR LIST/                      of that pair)
5 OUTPUT MATCH OF /ELEMENT/        (Otherwise, repeat the process
  AND BUTFIRST2 OF /PAIR LIST/     with the remaining pairs on the
END                                list)
```

This procedure uses the subprocedures SECOND and BUTFIRST2. The procedure SECOND outputs the second element of a specified list.

```
TO SECOND /LIST/
1 OUTPUT FIRST OF BUTFIRST OF /LIST/
END
```

The procedure BUTFIRST2 takes off the first two elements (i.e., the first pair of elements) from a list.

```
TO BUTFIRST2 /LIST/
1 OUTPUT BUTFIRST OF BUTFIRST OF /LIST/
END
```

Let's try MATCH.

```
←PRINT MATCH OF "X" AND "A 1 CAT K 3 BB X HOORAY T 27 X LOST"
HOORAY
←
```

The MATCH procedure outputs the second element associated with the first occurrence of the designated first element (in this case "X"). When the designated element is not present as the first element of some pair on the list, MATCH outputs the empty word, as follows.

```
←PRINT MATCH OF "BB" AND "A 1 CAT K 3 BB X HOORAY T 27 X LOST"

←
```

We might want MATCH to output *all* second elements of pairs starting with the designated element. We can write a procedure MATCHES to do this.

```
TO MATCHES /ELEMENT/ /PAIR LIST/
1 TEST EMPTYP /PAIR LIST/                (As before
2 IF TRUE OUTPUT /EMPTY/                  in
3 TEST IS /ELEMENT/ FIRST OF              MATCH)
  /PAIR LIST/
4 IF TRUE OUTPUT SENTENCE OF             (If a match is found, put the
  SECOND OF /PAIR LIST/                  associated second element onto
  MATCHES OF /ELEMENT/ AND               a list and continue looking
  BUTFIRST2 OF /PAIR LIST/               for further matches)
5 OUTPUT MATCHES /ELEMENT/               (Otherwise, continue looking
  BUTFIRST2 OF /PAIR LIST/               for further matches on the rest
END                                      of the pair list)
```

The procedure MATCHES forces the search to continue until the list is exhausted. Its effect is shown as follows.

```
←PRINT MATCHES OF "X" AND "A 1 CAT X 3 BB X HOORAY T 27 X LOST"
HOORAY LOST
←
```

2.1 Relations

We have been considering lists of *ordered* pairs of elements. If the positions of the two elements in each pair were reversed so that the second elements become first elements and vice versa, the results from using the procedures MATCH and MATCHES would have been different. Thus, for example,

```
←PRINT MATCH OF "X" AND "1 A X CAT BB 3 HOORAY X 27 T LOST X"
CAT
←
```

Sometimes we will want to switch the elements of a pair list in just that way, reversing first and second elements throughout. We can write a procedure SWITCH-PAIRS to do this operation for us.

```
TO SWITCH-PAIRS /PAIR LIST/
1 TEST EMPTYP /PAIR LIST/
2 IF TRUE OUTPUT /EMPTY/
3 OUTPUT SENTENCE OF
  (SENTENCE OF SECOND OF /PAIR LIST/ AND
  FIRST OF /PAIR LIST/)
  (SWITCH-PAIRS OF BUTFIRST2 OF /PAIR LIST/)
END
```

The procedure SWITCH-PAIRS forms a new pair list by reversing the two elements on the first pair of the input list, and doing the same thing on the rest of the input until it is exhausted. Its effect is clearly seen in the following examples.

```
←PRINT SWITCH-PAIRS "A 1 B 2 C 3 9 Z"
1 A 2 B 3 C Z 9
←PRINT MATCH OF "CAT" AND SWITCH-PAIRS OF "1 C 2 CA 3 CAT 4 CATS"
3
←
```

A set of ordered pairs is called a *relation*. The set we get by taking all the different first elements is called the *domain* of the relation; the set of its second elements is called the *range* of the relation. Thus, in the pair list "A 1 A 2 B 3 C 4 D 5 A 5" the domain is "A B C D" and the range is "1 2 3 4 5".

When we invert a relation, say by using the procedure SWITCH-PAIRS, its domain and range are interchanged as well. Thus, in the above example, SWITCH-PAIRS OF "A 1 A 2 B 3 C 4 D 5 A 5" is "1 A 2 A 3 B 4 C 5 D 5 A" and this relation has "1 2 3 4 5" as its domain and "A B C D" as its range.

To find the domain and range of a relation, we begin by extracting all first-elements and all second-elements as follows:

```
TO FIRST-ELEMENTS /PAIR LIST/
1 TEST EMPTYP /PAIR LIST/
2 IF TRUE OUTPUT /EMPTY/
3 OUTPUT SENTENCE OF
  (FIRST OF /PAIR LIST/)
  (FIRST-ELEMENTS OF BUTFIRST2 OF /PAIR LIST/)
END

TO SECOND-ELEMENTS /PAIR LIST/
1 TEST EMPTYP /PAIR LIST/
2 IF TRUE OUTPUT /EMPTY/
3 OUTPUT SENTENCE OF
  (SECOND OF /PAIR LIST/)
  (SECOND-ELEMENTS OF BUTFIRST2 OF /PAIR LIST/)
END
```

The two procedures are perfectly parallel. FIRST-ELEMENTS forms a list of all first elements of the input list; SECOND-ELEMENTS gives all second elements of the input list. The domain, however, is the list of all the *different* first elements, and the range is

the list of all different second elements on a list. We must therefore write a procedure REDUCE /LIST/ to eliminate repetitions on /LIST/. REDUCE uses a procedure FIND /ELEMENT/ /LIST/ which finds whether or not /ELEMENT/ appears on /LIST/. REDUCE /LIST/ simply uses FIND with each element on /LIST/ to see if that element occurs elsewhere on the list. If so, REDUCE ignores it; if not, it includes it in the output list it is making. We can write REDUCE as follows.

```
TO REDUCE /LIST/
1 TEST EMPTYP /LIST/                (When input list is exhausted,
2 IF TRUE OUTPUT /EMPTY/            terminate the procedure)
3 TEST FIND FIRST /LIST/            (If first element is repeated
  BUTFIRST /LIST/                   elsewhere on the list,)
4 IF TRUE OUTPUT REDUCE OF          (Ignore it and repeat process with
  BUTFIRST OF /LIST/                the rest of the list)
5 OUTPUT SENTENCE OF (FIRST         (Otherwise, put this element on the
  OF /LIST/) AND (REDUCE OF         output list and repeat the process
  BUTFIRST OF /LIST/)
END


TO FIND /ELEMENT/ /LIST/
1 TEST EMPTYP /LIST/                (If list is exhausted,)
2 IF TRUE OUTPUT "FALSE"            (element was not found)
3 TEST IS /ELEMENT/ FIRST OF
  /LIST/
4 IF TRUE OUTPUT "TRUE"             (Element is found if it is the
5 OUTPUT FIND /ELEMENT/             first thing on the list)
  (BUTFIRST OF /LIST/)              (Otherwise, repeat the process with
END                                 BUTFIRST of /LIST/)
```

Let's try the subprocedure FIND.

```
←PRINT FIND OF "B" "A B C B A"
TRUE
←PRINT FIND OF "D" "A B C B A"
FALSE
```

Now let's try REDUCE.

```
←PRINT REDUCE OF "A B C D E F"
A B C D E F
←PRINT REDUCE OF "A B C D A E F"
B C D A E F
```

REDUCE works properly but it has one aspect we might judge undesirable -- it does not keep the leftmost occurrence of any repeated elements on the original list. In the last example, we might have wished REDUCE to output "A B C D E F" instead of "B C D A E F". To fix this, we can rewrite REDUCE to search from right to left, and thus eliminate all but the leftmost occurrence of each element.

```
TO REDUCE /LIST/
1 TEST EMPTYP /LIST/
2 IF TRUE OUTPUT /EMPTY/
3 TEST FIND LAST OF /LIST/
    BUTLAST OF /LIST/
4 IF TRUE OUTPUT REDUCE OF BUTLAST OF /LIST/
5 OUTPUT SENTENCE OF
    (REDUCE OF BUTLAST OF /LIST/)
    (LAST OF /LIST/)
END
```

This new REDUCE is substantially the same as the old one except that it works on the input list from right to left, appending non-repeating elements onto the right of the output list it is constructing. It thereby preserves the "initial" ordering of repeated elements.

```
←PRINT REDUCE "A B A C B D Z D Y Z X"
A B C D Z Y X
```

We can now write a procedure DOMAIN /LIST/ simply by applying REDUCE to the output of FIRST-ELEMENTS /LIST/ and a procedure

RANGE by applying REDUCE to the output of SECOND-ELEMENTS.

```
TO DOMAIN /LIST/
1Ø OUTPUT REDUCE FIRST-ELEMENTS /LIST/
END

TO RANGE /LIST/
1Ø OUTPUT REDUCE SECOND-ELEMENTS /LIST/
END
```

The effect of the procedures DOMAIN and RANGE can now be illustrated in the following examples.

```
←PRINT DOMAIN OF "A AB B BOO C CAT D 1 A 2 C 3 AB 1 BOO 2"
A B C D AB BOO

←PRINT RANGE OF "A AB B BOO C CAT D 1 A 2 C 3 AB 1 BOO 2"
AB BOO CAT 1 2 3
```

The operations DOMAIN, RANGE, and SWITCH-PAIRS can be combined in various ways, for example:

```
←PRINT DOMAIN OF SWITCH-PAIRS OF "A AB B BOO C CAT D 1 A 2 C 3
  AB 1 BOO 2"
AB BOO CAT 1 2 3
```

2.2 Graphing

A relation is often more easily described and comprehended in a graphic representation than as a list or table. Given a relation, expressed in the form of a list of ordered pairs, we can write a procedure PLOT which prints out a graph of the relation. There are several ways of doing this, depending upon the convention chosen for representing the axes, heading formats for the domain and range elements, plotting characters, and the like. To get started, we will write a PLOT procedure whose effect is illustrated as follows.

```
PLOT "A 1 B 2 C 2 D 3 E 3 F 4 G 4 H 5 H 3 H 1"

  ABCDEFGH
1 .       .
2  ..
3    ..   .
4      ..
5        .
←
```

PLOT first types out a heading line, the horizontal axis, comprising the elements in the *domain* of its input list. It then plots a line for each element in the *range* of its input list as follows. It starts the line off by typing the element itself, to label the line. Each successive position in that line now represents an ordered pair; its column heading labels its domain element and its row heading its range element. The position is typed with a "." or a blank space according to whether or not that pair actually is contained in the input list. We assume that the domain and range elements are single characters; the extension to elements of larger and varying size is straightforward.

The procedure PLOT /PAIR LIST/ starts the graphing by finding the range and domain (RANGE OF /PAIR LIST/ and DOMAIN OF /PAIR LIST/), typing the title line for the horizontal axis, and then calling on the subprocedure PLOTLINES to carry out the graphing, line by line.

```
TO PLOT /PAIR LIST/
1 MAKE "RANGE" RANGE OF /PAIR LIST/
2 MAKE "DOMAIN" DOMAIN OF /PAIR LIST/
3 TYPE /BLANK/                          (Indent one space)
4 TYPE COMPACT OF /DOMAIN/              (Type horizontal axis title)
5 PLOTLINES /PAIR LIST/ /RANGE/ /DOMAIN/ (Plot all the lines)
END
```

where the procedure COMPACT OF /LIST/ removes all spaces in /LIST/, i.e., makes it compact.

```
TO COMPACT /LIST/
1 TEST EMPTYP /LIST/
2 IF TRUE OUTPUT /EMPTY/
3 OUTPUT WORD OF (FIRST OF /LIST/)
  (COMPACT BUTFIRST /LIST/)
END
```

The procedure PLOTLINES, in turn, merely labels each line of the graph with the next element in the range, calling on its sub-procedure PLOTPOINTS to do the actual plotting of points for the line.

```
TO PLOTLINES /PAIR LIST/ /RANGE/ /DOMAIN/
1 TEST EMPTYP /RANGE/              (If the range list is exhausted,)
2 IF TRUE STOP                     (The process terminates)
3 TYPE /SKIP/                      (Go to next row)
4 TYPE FIRST OF /RANGE/            (Type the current range element)
5 PLOTPOINTS /RANGE/ /DOMAIN/      (Plot all points in the pair list
                                    having this current range element)
6 PLOTLINES /PAIR LIST/            (Continue the process with the rest
  (BUTFIRST OF /RANGE/) /DOMAIN/   of the elements in the range list)
END
```

PLOTPOINTS has two inputs, the lists /D/ and /R/. It checks for each successive element in /D/ whether the pair formed by that element and the first element in /R/ is in the original /PAIR LIST/. If it is, it types a mark ("."); if not, it skips a space.

```
TO PLOTPOINTS /R/ /D/
1 TEST EMPTYP /D/                  (If /D/ is exhausted,)
2 IF TRUE STOP                     (terminate process)
3 TEST CONTAINSP                       (Is the pair comprising the
  SENTENCE (FIRST /D/) (FIRST /R/)     first elements in /D/ and /R/
  /PAIR LIST/                          on the /PAIR LIST/)
4 IF TRUE TYPE "."
5 IF FALSE TYPE /BLANK/
6 PLOTPOINTS /R/ (BUTFIRST /D/) (Continue process with rest of /D/)
END
```

PLOTPOINTS itself uses a subprocedure CONTAINSP /PAIR/ /PAIR LIST/ which outputs "TRUE" if the pair /PAIR/ is on /PAIR LIST/ and "FALSE" otherwise.

```
TO CONTAINSP /PAIR/ /PAIR LIST/
1 TEST EMPTYP /PAIR LIST/          (If list is exhausted)
2 IF TRUE OUTPUT "FALSE"           (It did not contain the pair)
3 TEST IS /PAIR/ (FIRST2 OF        (If the pair is at the head of
  /PAIR LIST/)                     the list)
4 IF TRUE OUTPUT "TRUE"            (It is contained in the list)
5 OUTPUT CONTAINSP /PAIR/          (Otherwise, look for the pair in
  (BUTFIRST2 OF /PAIR LIST/)       the rest of the list)
END
```

CONTAINSP uses FIRST2 OF /LIST/ which outputs the first pair of its input list.

```
TO FIRST2 /LIST/
1 OUTPUT SENTENCE OF (FIRST OF /LIST/) (SECOND OF /LIST/)
END
```

When PLOT OF /PAIR LIST/ is used to make a graph, it labels the domain row and range column, using the elements of each in the order in which they appear on /PAIR LIST/. And, this might not be the order we desire. For example,

```
PLOT "H 2 E 1 E 3 X Ø X 4 A Ø A 4 G Ø G 4 O 1 O 3 N 2"
```

```
  HEXAGON
2 .     .
1  .   .
3  .   .
Ø   ...
4   ...
```

produces a graph which has apparently been mistitled. Using the same pair list, if we had reordered the elements in the range as shown in the following, we would have had this rather different result.

```
  HEXAGON
Ø   ...
1  .   .
2 .     .
3  .   .
4   ...
```

There is an obvious way of gaining control over the plotting procedure to permit us to graph relations using given orderings of the elements. All we need do is explicitly specify the domain and range lists, with the elements ordered as we wish. Thus, we can write a new graphing procedure, PRETTYPLOT, which has two inputs in addition to the original pair list -- a list of domain elements and a list of range elements, ordered any way we like.

```
TO PRETTYPLOT /PAIR LIST/ /RANGE/ /DOMAIN/
1 TYPE /BLANK/
2 TYPE COMPACT OF /DOMAIN/
3 PLOTLINES /PAIR LIST/ /RANGE/ /DOMAIN/
END
```

PRETTYPLOT is the same procedure as PLOT except that it incorporates two extra inputs instead of generating range and domain from the /PAIR LIST/. It uses the same subprocedure PLOTLINES. But the command

```
PRETTYPLOT "H 2 E 1 E 3 X Ø X 4 A Ø A 4 G Ø G 4 O 1 O 3 N 2"
  "Ø 1 2 3 4" "HEXAGON"
```

will produce the graph just shown (and there is no way of reordering /PAIR LIST/ to accomplish the same result using PLOT).

Both PLOT and PRETTYPLOT type out a horizontal list of numbers, labeling the first elements of the pairs being plotted, and a vertical list, labeling the second elements. These identify the pairs in the graph. Usually, one plots a graph to gain some geometric insights into the relation between the various pairs, when the first or second element is changed in a systematic way. For example, we might ask whether the size of the second element increases or decreases as larger and larger values of the first element are taken. To facilitate this sort of inquiry on sets of number pairs, we adopt two plotting conventions. First, we write the vertical and horizontal labeling lists in order of size.

Also, to get some feeling for sizes of numbers directly from a graph, we label the elements along the axes using a uniform increment. We can do this for pairs having integer elements simply by filling in all the missing integers in the labeling lists. The procedure FILLIN performs these two operations on lists with integer elements, as follows.

FILLIN finds the largest numerical element in a list if the list contains all numbers (else it outputs the original list). Then it generates an output list whose elements are the successive integers from Ø to this largest number.

```
TO FILLIN /LIST/
1 MAKE "LARGEST" Ø                    (Assume the largest element is zero)
2 TEST ALLNUMBERS /LIST/              (Are all elements in the list numbers?)
3 IF FALSE OUTPUT /LIST/              (If not, output the list)
4 OUTPUT NUMBERLIST OF /LARGEST/      (Otherwise, generate the output
END                                    list)
```

The procedure ALLNUMBERS /LIST/ as well as testing all elements in /LIST/ to see if they are numbers, also finds the maximum element along the way.

```
TO ALLNUMBERS /LIST/
1 TEST EMPTYP OF /LIST/               (If list is exhausted,)
2 IF TRUE OUTPUT "TRUE"               (it contained only numbers)
3 TEST NUMBERP OF (FIRST OF           (Is first element a number?)
  /LIST/)                             (If not, list contains a non-
4 IF FALSE OUTPUT "FALSE"             number)
5 MAKE
    NAME:  "LARGEST"                                (Compare largest element
    THING:  MAXIMUM OF (FIRST OF /LIST/) so far encountered with
            AND /LARGEST/                            first on list)
6 OUTPUT ALLNUMBERS OF                (Continue process with remaining
  (BUTFIRST OF /LIST/)                elements in list)
END
```

The procedure for generating a list of the integers from Ø to /NUMBER/ is,

```
TO NUMBERLIST /NUMBER/
1 TEST IS /NUMBER/ Ø
2 IF TRUE OUTPUT Ø
3 OUTPUT SENTENCE OF
  NUMBERLIST (DIFFERENCE OF /NUMBER/ AND 1)
  /NUMBER/
END
```

Let's try these procedures singly, and then, together to show their usefulness in graphing.

```
←PRINT NUMBERLIST OF 12
Ø 1 2 3 4 5 6 7 8 9 1Ø 11 12
←PRINT ALLNUMBERS OF "H E X A G O N"
FALSE
←PRINT ALLNUMBERS OF "1 9 25 16 4"
TRUE
```

```
←PRINT FILLIN OF "5 4 9 3 1 ANY NON-NUMBER STUFF"
5 4 9 3 1 ANY NON-NUMBER STUFF
←PRINT FILLIN OF "5 4 9 3 1"
Ø 1 2 3 4 5 6 7 8 9
←MAKE "PAIR LIST" "2 4 7 9 3 1 5 1 4 Ø 6 4 1 9"
←PLOT /PAIR LIST/


  2735461
4 .     .
9  .     .
1    ..
Ø      .
```

```
←PRETTYPRINT /PAIR LIST/ (FILLIN OF RANGE OF /PAIR LIST/)
  (FILLIN OF DOMAIN OF /PAIR LIST/)


  1234567
Ø     .
1    . .
2
3
4  .    .
5
6
7
8
9 .      .
```

This use of PRETTYPRINT together with FILLIN shows a "simple" graph that was obscured when PLOT was used.

The procedures PLOT and PRETTYPLOT each make a graph corresponding to a given list of pairs. Sometimes we wish to go the other way, i.e., to input a graph and generate a corresponding list of pairs as an output. We write next a set of procedures to do this. Once again the main procedure, PICTURELIST in this instance, uses a subprocedure which uses a subprocedure. The procedure PICTURELIST has no inputs. It accepts a graph as it is typed in by the user in the same form as those generated by PLOT. After each line is entered, the points are translated to pairs and appended to the list being generated. We can start by using an auxiliary procedure, INSTRUCT, to give instructions to the user.

```
TO INSTRUCT
1 PRINT "FIRST LIST THE ELEMENTS IN THE DOMAIN, THEN ENTER
  THE POINTS, LINE BY LINE.  BEGIN EACH LINE WITH A LABEL
  FOR THAT RANGE ELEMENT, THEN TYPE DOTS UNDER THE DOMAIN
  ELEMENTS WHICH ARE THE FIRST ELEMENTS OF THE PAIRS TO BE
  INCLUDED.  AFTER ENTERING LAST LINE, TYPE IN ↑ TO TERMINATE"
2 OUTPUT PICTURELIST
END
```

Next we give the main procedure, which makes sure that the list to be constructed starts off being empty, requests the horizontal heading (the domain elements) from the user and then calls a subprocedure to do the real work.

```
TO PICTURELIST
1 TYPE /BLANK/
2 TYPE /BLANK/                          (Indent two spaces)
3 MAKE "PAIR LIST" /EMPTY/              (Make sure that pair list is empty
                                        to start with)
4 MAKE "DOMAIN" REQUEST                 (Wait for user to type in a word
                                        listing elements in domain)
5 OUTPUT CONSTRUCT OF /PAIR LIST/  (Construct output list)
END
```

The subprocedure CONSTRUCT requests the graphical data, line by line, from the user. The first character on each line is either ↑ or a range element. The succeeding characters are dots and blanks. The subprocedure ENTERPAIRS converts the points on each successive line to pairs and appends these to /PAIR LIST/.

```
TO CONSTRUCT /PAIR LIST/
1 MAKE "ROW" REQUEST                    (Wait for user to type in a line)
2 TEST IS (FIRST OF /ROW/) "↑"          (If last line was typed in)
3 IF TRUE OUTPUT /PAIR LIST/            (terminate process)
4 ENTERPAIRS (BUTFIRST OF /ROW/)        (Otherwise, convert points in
  /DOMAIN/                               current line to pairs on output list)
5 OUTPUT CONSTRUCT OF /PAIR LIST/       (and continue process with next
END                                      line)
```

The lowest subprocedure in the program structure, ENTERPAIRS, is called to process the points in a line just after the line is typed in by the user. This procedure builds up the list which is to be the output, /PAIR LIST/, by adding to it a pair of elements for each point of the graph. ENTERPAIRS has two inputs, a word /LINE/ comprising marks and blank spaces (indicating points and non-points in the current line) and the word of domain elements (DOMAIN).

```
TO ENTERPAIRS /LINE/ /DOMAIN/
1 TEST EMPTYP /LINE/                    (If all the points are processed)
2 IF TRUE STOP                          (terminate)
3 TEST IS (FIRST OF /LINE/) /BLANK/
4 IF FALSE MAKE
    NAME:  "PAIR LIST"                  (If the first element of
    THING:  SENTENCE OF SENTENCE OF      /LINE/ is a datum point,
            /PAIR LIST/                  tack on to /PAIR LIST/ the
            FIRST OF /DOMAIN/            designated pair, labeled by
            FIRST OF /ROW/               the first elements of /DOMAIN/
                                         and /ROW/)
5 ENTERPAIRS (BUTFIRST OF /LINE/)       (Repeat the process with the
  (BUTFIRST OF /DOMAIN/)                 next positions of /LINE/ and
END                                      /DOMAIN/)
```

Here is an illustration of the use of PICTURELIST.

```
←PRINT PICTURELIST

  ABCDEFG                          (User types in domain elements)
*Ø    .                           (and then successive lines,
*1   ...                           one for each range element)
*2 .....                         (The asterisk indicates that LOGO
*3.......                        is ready for the next input)
*4 .....
*5   ...
*6    .
*↑                               (The user's ↑ terminates the input)

D Ø C 1 D 1 E 1 B 2 C 2 D 2 E 2 F 2 A 3 B 3 C 3 D 3 E 3 F 3 G 3
B 4 C 4 D 4 E 4 F 4 C 5 D 5 E 5 D 6
```

The output of PICTURELIST is a list of pairs and this list can be directly used as an input with procedures we have previously written. Thus, for example, we can use SWITCH-PAIRS to reverse the list output by PICTURELIST and then use PLOT to graph this reversed list.

```
←PLOT SWITCH-PAIRS OF PICTURELIST

  REFLXNS                          (The user inputs this graph)
*M.......
*O .    .
*N  . .
*S   .
*T  . .
*E .    .
*R.      .
*↑


 MONSTER                           (PLOT outputs this graph)
R.      .
E..    .
F. . .
L.  .
X. . .
N..    .
S.      .
←
```

2.3 Functions

Some operations on pair lists, or on their graphs, seem to change them radically, and others only slightly and predictably. For example, the effect of SWITCH-PAIRS on a pair list is simply to reverse the order of each pair and in the case of the above example the graph of the switched relation is visibly similar to the graph of the original we typed in to PICTURELIST. The new graph is obviously a reflection of the original one about the diagonal. This, however, is deceptive. If we had omitted the interior points on the first line of the original graph and had instead typed in

```
  REFLXNS
*M.     .
*O .   .
*N  . .
*S   .
*T  . .
*E .   .
*R.     .
*↑
```

Then the result of PLOT OF SWITCH-PAIRS OF PICTURELIST would have been the graph:

```
 MONSTER
R.     .
S.     .
E .   .
N .   .
F  . .
X  . .
L   .
```

And this jagged V-shaped object is hardly a reflection of its X-shaped ancestor.

In point of fact, we did not offer any reason for expecting the operation SWITCH-PAIRS to have the effect of a reflection. In order to have such an effect, SWITCH-PAIRS would have to preserve the original order of the elements in the domain of its input list, i.e., the elements of its range would need to be plotted in the order REFLXNS. SWITCH-PAIRS preserves the order of the range elements of its input list (these become the domain elements of its output). But it changes the order of the domain elements of its input list (these become the range elements of its output). We can reorder the list of range elements before plotting them and thereby not obscure its reflection effect. (In fact, for numerical elements we have the procedure ORDER for doing this.) But this situation suggests a more general question -- if two pair lists are not identical (or if their graphs are not identical), might they nevertheless represent the same relation?

The operation SWITCH-PAIRS takes a list representing a relation and outputs a list representing a related, though distinctly different, relation. There are some operations on relations which leave them unchanged, i.e., where the output list and input list represent the same relation. For example, the operation of reversing the order among pairs (instead of within pairs, as in SWITCH-PAIRS) preserves the original relation. Consider the following two lists (call them /LIST 1/ and /LIST 2/:
"Ø -1 Ø 1 1 Ø 1 2 2 1 2 3 3 2 3 4" and
"3 4 3 2 2 3 2 1 1 2 1 Ø Ø 1 Ø -1".

These lists contain the same pairs, though written down in different order. Clearly these two lists are two of several representations of the same relation. Nevertheless, matching operations on /LIST 1/ and /LIST 2/ generally produce different results, as shown in the following examples.

```
←MATCH "1" /LIST 1/
Ø
←MATCH "1" /LIST 2/
2

←MATCHES "3" /LIST 1/
2 4
←MATCHES "3" /LIST 2/
4 2
```

We might reasonably ask whether there are relations for which all rearrangements of the orderings among pairs of elements have the results of the operations MATCH and MATCHES invariant. Clearly this cannot be true for relations where any element in the domain has more than one element associated with it. Relations in which each element of the domain occurs as the first element of no more than one pair are called *functions*.

We can easily write a procedure FUNCTIONP to determine whether or not a pair list represents a function. FUNCTIONP simply tries to match the first element (that is, the domain element) of each pair in the list with the rest of the list. If it is unable to do so for any first element, the pair list represents a function. Otherwise, it is not.

```
TO FUNCTIONP /PAIR LIST/
1 TEST EMPTYP /PAIR LIST/          (If list becomes exhausted)
2 IF TRUE OUTPUT "TRUE"            (it represents a function)
3 TEST EMPTYP MATCH                (Does the first element of the first
  (FIRST OF /PAIR LIST/)           pair match any other pair in the
  (BUTFIRST2 OF /PAIR LIST/)       list?)
4 IF FALSE OUTPUT "FALSE"          (If so, the list is not a function)
5 OUTPUT FUNCTIONP (BUTFIRST2      (If not, repeat the test with the
  OF /PAIR LIST/)                  remaining pairs of the list)
END
```

The effect of FUNCTIONP is illustrated by

```
←PRINT FUNCTIONP "A 1 B 2 C 3 D 4 E 5 F 6 G 7 H 8 I 9"
TRUE
←PRINT FUNCTIONP "A 1 E 1 I 1 O 1 U 1"
TRUE
←PRINT FUNCTIONP "1 A 1 E 1 O 1 I 1 U"
FALSE
```

A function is a relation where a given element of the domain matches a unique element of the range. Given an input to match, there is a single output, regardless of how the list of pairs is ordered. Why should one care about this kind of singular definiteness in an answer? Using the operation MATCHES, we can find all elements in the range that match a given element in the domain, so what advantage is there if there is only one such element instead of, say, two?

One way of showing the usefulness of this property comes from considering the operation of *composing* two pair lists. The output from composing two pair lists is another pair list each of whose pairs consists of a domain element /D1/ from the first list and a range element /R2/ from the second list such that MATCH OF /R1/ AND /LIST 2/ gives the output /R2/. (/R1/ is the range element paired with /D1/.) For example,

```
←COMPOSE "A 1 B 2 C 3 D 4 E 5" AND "1 Z 3 Y 2 X"
A Z B X C Y
←
```

The procedure follows the description just given.

```
TO COMPOSE /PAIR LIST 1/ /PAIR LIST 2/
1Ø TEST EMPTYP /PAIR LIST 1/    (When the first list is exhausted)
2Ø IF TRUE OUTPUT /EMPTY/       (terminate the process)
3Ø MAKE
    NAME:  "R2"
    THING:  MATCH OF (SECOND OF /PAIR LIST 1/)
            AND /PAIR LIST 2/
4Ø TEST IS /R2/ /EMPTY/         (Does the second element of the
                                first list fail to match a first
                                element of the second list?)
```

```
5Ø IF TRUE OUTPUT COMPOSE OF   (If so, try matching the other
   (BUTFIRST2 OF /PAIR LIST 1/)  elements of the first list with
   AND /PAIR LIST 2/             the second list)
6Ø OUTPUT SENTENCE OF SENTENCE OF
   FIRST OF /PAIR LIST 1/                (Add matched pair to
   /R2/                                  output list being
   COMPOSE (BUTFIRST2 /PAIR LIST 1/)     generated, and continue)
   /PAIR LIST 2/
END
```

When its two inputs are *functions*, COMPOSE outputs a function list whose pairs do not change (though possibly they occur in different orders) when the order of pairs within either of the two input lists is changed. And this is true only if the inputs are functions, not other relations. Otherwise, for example,

```
←PRINT COMPOSE OF "A 1 B 2 C 3"  "1 Z 2 X 1 Y"
A Z B X
```

At the same time, however,

```
←PRINT COMPOSE OF "A 1 B 2 C 3" "1 Y 2 X 1 Z"
A Y B X
```

Even though "1 Z 2 X 1 Y" and "1 Y 2 X 1 Z" represent the same relation, COMPOSE gives different results depending upon which representation is used. A similar problem exists with the *first* list when it is not a function.

To obtain unambiguous results, we shall require the two inputs to be functions, by adding the following two lines to COMPOSE.

```
1 TEST BOTH FUNCTIONP /PAIR LIST 1/ AND FUNCTIONP /PAIR LIST 2/
2 IF FALSE OUTPUT /EMPTY/       (Terminate if either input is not
                                 a function)
```

With this modification, the operation of composing two lists will always produce a unique output list (this will be the empty list, if the inputs are not functions). The output is dependent on the order of the two inputs as shown in the following example:

```
←PRINT COMPOSE OF "A 1 B 2 C 3" AND "1 C 2 B 3 A"
A C B B C A
←PRINT COMPOSE OF "1 C 2 B 3 A" AND "A 1 B 2 C 3"
1 3 2 2 3 1
```

Though the input lists are the same in both cases, the two output lists do not even have the same elements.

In general, then, COMPOSE OF /LIST 1/ AND /LIST 2/ and COMPOSE OF /LIST 2/ AND /LIST 1/ are different lists, so COMPOSE is not a commutative operation. On the other hand, COMPOSE is an associative operation. COMPOSE OF /LIST 1/ AND (COMPOSE OF /LIST 2/ AND /LIST 3/) generates the same output list as COMPOSE OF (COMPOSE OF /LIST 1/ AND /LIST 2/) AND /LIST 3/. That is, the order of composition does not affect the outcome, given that the order of the input lists is not changed.

2.4 Coding and Decoding

The operations of matching and of composing functions are useful in coding messages, for example in letter substitution ciphers. To illustrate, let's make up two such ciphers, in the form of pair lists. The first one, "SUCCESSOR" associates each letter in the English alphabet with the letter following.

```
MAKE
   NAME:  "SUCCESSOR"
   THING:  "A B B C C D D E ... X Y Y Z Z A"
```

where "A" is taken as the successor of "Z".

The second substitution list, "OPPOSITE", associates the Nth letter in the alphabet counting from left to right with the Nth letter counting from right to left. Thus, "A" is associated with "Z", "B" with "Y", and so on as follows.

```
MAKE
   NAME:  "OPPOSITE"
   THING:  "A Z B Y C X ... X C Y B Z A"
```

Using the procedure MATCH with /SUCCESSOR/ as input we can encode any letter, for example "C", as follows.

```
←PRINT MATCH OF "C" AND /SUCCESSOR/
D
```

By doing this successively, letter by letter, we can translate a word, say "CAT" into the word "DBU". Similarly, using MATCH with the substitution list /OPPOSITE/, we would translate "CAT" to "XZG". In general, we can write a procedure CODEWORD which takes two inputs, the word to be translated, and the pair list comprising the letter substitution code. CODEWORD outputs the encoded word.

```
TO CODEWORD /WORD/ /LIST/
1 TEST EMPTYP /WORD/              (When entire word is translated,)
2 IF TRUE OUTPUT /EMPTY/          (Terminate)
3 OUTPUT WORD OF
   MATCH (FIRST OF /WORD/) AND /LIST/
   CODEWORD (BUTFIRST /WORD/) AND /LIST/
END
```

The last line of CODEWORD forms the answer one letter at a time by matching successive letters of /WORD/.

```
←PRINT CODEWORD OF "CAT" /SUCCESSOR/
DBU
←PRINT CODEWORD OF "CAT" /OPPOSITE/
XZG
```

We can compose a substitution code from existing codes, for example as follows.

```
←PRINT CODEWORD OF "CAT" AND (COMPOSE OF /SUCCESSOR/ AND /OPPOSITE/)
WYF
←PRINT CODEWORD OF "CAT" AND (COMPOSE OF /OPPOSITE/ AND /SUCCESSOR/)
YAH
←
```

We can extend our translation to include whole sentences by writing a procedure ENCODE which uses CODEWORD as a subprocedure. ENCODE takes two inputs, the sentence to be translated and, once again, the letter substitution code.

```
TO ENCODE /MESSAGE/ /LIST/
1 TEST EMPTYP /MESSAGE/              (When entire sentence is translated)
2 IF TRUE OUTPUT /EMPTY/             (Terminate)
3 OUTPUT SENTENCE OF
     CODEWORD OF (FIRST OF /MESSAGE/) /LIST/
     ENCODE (BUTFIRST OF /MESSAGE/) /LIST/
END
```

ENCODE simply uses CODEWORD on each word of the message.

```
←PRINT ENCODE OF "THIS IS SECRET" AND /OPPOSITE/
GSRH RH HVXIVG
←PRINT ENCODE "THIS IS SECRET" (COMPOSE OF /OPPOSITE/ AND /SUCCESSOR/)
HTSI SI IWYJWH
←
```

Using ENCODE and COMPOSE we can "hide" messages in a variety of ways. Given some such encoded message, how should we go about trying to decode it? Assuming we know that it was encoded by a letter substitution cipher or a composition of such ciphers, we might systematically go about trying all possible coding functions of that kind and, since there are a finite number, we would eventually find (at least one) readable English message.

Rather than ask how to devise good systematic approaches in this blind case where we do not know which code was used, let's begin with the much simpler situation where we do know the code. All we need do, then, is to write a decoding procedure which given the coded message as an input, along with the substitution code used for encoding it, gives back the original message.

Say that we wish to decode a message, that is, we wish to substitute for each letter in the coded message its uncoded match. It is clear that decoding is itself a coding problem -- the only difference between coding and uncoding a message is that different substitution lists are to be used. The easiest way to find the substitution list to be used for decoding is to note that we now want to use the original coding list "backwards". We want to find the first element corresponding to a given letter in the range of the coding relation. Thus, the substitution list we should use for decoding is just SWITCH-PAIRS of the original coding list, and we can write

```
TO DECODE /MESSAGE/ /CODE/
1 OUTPUT ENCODE OF /MESSAGE/ AND
  SWITCH-PAIRS OF /CODE/
END
```

Now we can unscramble coded sentences.

```
←PRINT DECODE OF "UIJT JT B TFDSFU DPEF" /SUCCESSOR/
THIS IS A SECRET CODE
```

DECODE also enables us to unconfound messages generated by composition of codes.

```
←PRINT DECODE OF "TME NME UJSN YAH" (COMPOSE OF /OPPOSITE/ AND
  /SUCCESSOR/)
HOW NOW GRIN CAT
```

Will our encoding and decoding procedures work for all letter substitution codes? Given any sentence /SENTENCE/ whose words are composed of characters in the domain of a pair list /CODE/, ENCODE /SENTENCE/ /CODE/ will surely generate a sentence output. Consider, however, this perfectly valid substitution code:

```
MAKE
   NAME:  "I"
   THING:  "A I B I C I D I ... X I Y I Z I"
```

```
←PRINT ENCODE OF "LOSING MY IDENTITY" /I/
IIIIII II IIIIIIII
```

Somehow this code does not seem to be a useful one. The reason is easy to show. Let us try to decode messages encoded by /I/.

```
←PRINT DECODE OF "IIIIII II IIIIIIII" /I/
AAAAAA AA AAAAAAAA
```

Although DECODE gave us an answer, it was not the message we started with. We have lost the original message beyond recall. The difficulty is clear enough. Though the pair list /I/ is a function and therefore yields an unequivocal output for each input in its domain, this output is the same for more than one of the inputs -- in fact, it is the same for all inputs. In consequence, given an output of /I/, we cannot determine which input actually gave rise to it. Put more succinctly, the pair list /I/ is not a good code, in the sense that any messages encoded by it are restorable by decoding, because its inverse pair list, SWITCH-PAIRS OF /I/, is not a function.

Let's incorporate in ENCODE and DECODE the requirement that the input /CODE/, and its inverse, SWITCH-PAIRS /CODE/, are both functions. (It is noteworthy that this requirement already exists for ENCODE whenever we use a composition of two codes, since COMPOSE requires that its two inputs are functions.) To begin with, we write a procedure INVERSEP which takes a pair list as an input and outputs "TRUE" or "FALSE" according as the inverse of the input (SWITCH-PAIRS of the input) is or is not a function.

```
TO INVERSEP /PAIR LIST/
1 TEST FUNCTIONP OF SWITCH-PAIRS OF /PAIR LIST/
2 IF TRUE OUTPUT "TRUE"
3 OUTPUT "FALSE"
END
```

Now we can test whether a proposed code is a good one by using the procedure TESTCODE.

```
TO TESTCODE /PAIR LIST/
1 OUTPUT BOTH FUNCTIONP /PAIR LIST/ AND INVERSEP /PAIR LIST/
END
```

INVERSEP, and therefore TESTCODE, would have ruled out /I/ as a useful code.

```
←PRINT INVERSEP /I/
FALSE
```

But they would have accepted /SUCCESSOR/.

```
←PRINT TESTCODE /SUCCESSOR/
TRUE
```

TESTCODE tests whether or not a code will preserve messages under composition of encoding and decoding operations, in either order. That is, TESTCODE OF /CODE/ is "TRUE" implies that ENCODE OF (DECODE OF /MESSAGE/ /CODE/) AND /CODE/ and DECODE OF (ENCODE OF /MESSAGE/ /CODE/) AND /CODE/ will both output /MESSAGE/.

Since we can certify whether any proposed pair list is useful for encipherment and decipherment of messages, we can try to devise a variety of different codes and study their operation. In doing so, we probably would note the limitations of pair lists as a means of expressing coding functions. For example, consider two of our existing codes /SUCCESSOR/ and /OPPOSITE/. We might like to modify these so that, instead of being defined absolutely (where a given letter is always associated with another given letter) they are defined relative to the message text being encoded or decoded.

Thus, we might want a code /NEXT/ similar to /SUCCESSOR/ but which encodes "CAT" as "ATC" (rather than "DBU" as /SUCCESSOR/ does) i.e., each letter is replaced by its successor in the given

message text, rather than by its successor in a fixed text, the English alphabet. To get back a word encoded by /NEXT/ we would, of course, replace each letter in the word by its predecessor so "ATC" would be decoded to "CAT". Thus, /NEXT/ would have a well-defined inverse.

The same kind of extension can be made with /OPPOSITE/. We wish to have a coding function, say /REVERSE/, which encodes "CAT" as "TAC". The function /REVERSE/ replaced the Nth letter from the left by its opposite, the Nth letter from the right, using the word itself, rather than a fixed list. And /REVERSE/ has an inverse function which turns out to be /REVERSE/ itself.

This sort of coding seems very similar to the kind done before, but in a way it is very different. We are no longer really coding letter by letter as before, but instead the coding depends on the rest of the message. For example, the "A" in "CAT" gets coded as "T" using /NEXT/ but the letter "A" in "CAB" gets coded as "B". Nevertheless, each message has a unique encoding using /NEXT/, and furthermore this encoding can be decoded. Here, however, we must have a list of pairs, the first element of which is a *message* and the second element the coding of that *message*, since we can no longer deal with letters as isolated objects. To actually write out /NEXT/, therefore, would be extremely tedious even if we restricted ourselves to a domain consisting of short messages. The task would be impossible if we wanted to write the pair list /NEXT/ to translate all possible messages. This seems strange since it is so easy to represent this coding process by a rule.

/NEXT/ and /REVERSE/ are more naturally expressed as LOGO procedures than as ordered lists of pairs. Their operations are much more easily described in the way used with MATCH or COMPOSE

even though their effects are more similar to those obtained with the two functions /SUCCESSOR/ and /OPPOSITE/. Up to now we have represented functions as pair lists and have written some LOGO procedures to facilitate the manipulation of these functions. We shall next consider the use of LOGO procedures as a primary means of representing functions.

3. Functions Specified by Rules

Thus far we have specified a function in a very literal fashion, by actually listing all its elements. Instead, we can specify functions by appropriate *rules*. These are rules which describe a process for effectively determining an element of the range set corresponding to any element of the domain set. There are various ways of expressing such rules. One familiar representation, which suffices for many numerical functions, is algebraic formulas. We shall use a more general representation, LOGO procedures.

3.1 Functions as Procedures

To start off, let's show how a LOGO procedure can be written to represent any function or relation which can be expressed as a list of ordered pairs. Here, for example, is a procedure corresponding to the pair list /SUCCESSOR/ discussed in Section 2.4.

```
TO SUCCESSOR /LETTER/
1 TEST IS /LETTER/ "A"
2 IF TRUE OUTPUT "B"
3 TEST IS /LETTER/ "B"
4 IF TRUE OUTPUT "C"
   : : :
51 TEST IS /LETTER/ "Z"
52 IF TRUE OUTPUT "A"
END
```

With the pair list representation, we used the procedure MATCH to find the range element associated with some given domain element, as follows.

```
←PRINT MATCH OF "X" AND /SUCCESSOR/
Y
```

The corresponding operation using the LOGO procedure for the successor function is:

```
←PRINT SUCCESSOR OF "X"
Y
```

We can modify SUCCESSOR to take care of inputs that are not allowed as elements in the domain (i.e., non-letters) by adding an instruction at the end of the procedure, as follows.

```
EDIT SUCCESSOR
53 OUTPUT SENTENCE OF /LETTER/ AND "IS NOT IN MY DOMAIN"
END
```

Now any input that is not matched in one of the 26 TEST instructions will cause this last instruction to be executed, for example:

```
←PRINT SUCCESSOR OF "1"
1 IS NOT IN MY DOMAIN
←
```

As well as encompassing all pair lists, LOGO procedures can be written to express functions whose elements cannot be listed exhaustively. Thus, a procedure for doubling any integer is:

```
TO DOUBLE /N/
1 TEST NUMBERP OF /N/                          (Is /N/ an integer?)
2 IF TRUE OUTPUT PRODUCT OF 2 AND /N/
3 OUTPUT SENTENCE OF /N/ AND "IS NOT AN INTEGER"
END

←PRINT DOUBLE OF "5"
1Ø
←PRINT DOUBLE OF 999999999999999
1999999999999998
←PRINT DOUBLE OF "A"
A IS NOT AN INTEGER
```

Some functions are defined over domains that include non-numbers as well as numbers. Thus,

```
TO SECOND /ANYTHING/
1 OUTPUT FIRST OF BUTFIRST OF /ANYTHING/
END

←PRINT SECOND OF "ABC"
B
←PRINT SECOND OF "123"
2
```

A LOGO function has an output. Thus, it can be used as an input to another function, whose output can again be another function's input. Thus, composition of functions is easily expressed by chaining procedures. For example,

```
←PRINT DOUBLE OF DOUBLE OF 5
2Ø
←PRINT SECOND OF DOUBLE OF SECOND OF 99
8
←PRINT DOUBLE OF QUOTIENT OF 35 AND 5
14
```

In the last example, the procedure DOUBLE was composed with the "built-in" LOGO function QUOTIENT. Any number of LOGO functions, whether these are procedures written by a user or those already included in LOGO (such as SUM, PRODUCT, QUOTIENT, DIFFERENCE, FIRST, LAST, COUNT, etc.) can be chained in this way. The only requirements are that the inputs are of the proper number and type. Thus, in

```
←PRINT DOUBLE OF SUCCESSOR OF "A"
B IS NOT AN INTEGER
```

the output of SUCCESSOR was not of the required type (integer) needed for use as an input to DOUBLE.

Some LOGO procedures are not functions. In order for a LOGO procedure to properly express a function, it must have an output

for each allowable input. Moreover, for each input the output must always be the same, i.e., must be unique. There are simple, and even very useful, procedures for which this uniqueness does not hold. One such procedure is NOTFUNCTION which uses the built-in LOGO operation RANDOM. (RANDOM outputs a generally different pseudo-random digit each time it is used.)

```
TO NOTFUNCTION /ANYTHING/
1 OUTPUT RANDOM
END

←PRINT NOTFUNCTION OF 1
8
←PRINT NOTFUNCTION OF 1
7
←PRINT NOTFUNCTION OF 1
2
←PRINT NOTFUNCTION OF 1
7
```

The difficulty is *not* that NOTFUNCTION ignores its input, /ANYTHING/ -- we can write a procedure which ignores its input and yet expresses a function.

```
TO BORE /EVERYBODY/
1 OUTPUT "GETTING TIRED OF THIS?"
END

←PRINT BORE OF 1
GETTING TIRED OF THIS?
←PRINT BORE OF 29
GETTING TIRED OF THIS?
←PRINT BORE OF 1
GETTING TIRED OF THIS?
```

In this case, unlike NOTFUNCTION, the procedure always gives the same output for a given input. So it is a function, though not a very useful one, since it always gives the same output for *any* input.

Another funny function, related to BORE, is expressed by the following procedure, SAME OF /X/.

```
TO SAME /X/
1 OUTPUT /X/
END
```

SAME's output is always the same as its input.

```
←PRINT SAME OF 77
77
←PRINT SAME OF "HI THERE"
HI THERE
←PRINT SAME OF SAME OF "HI THERE"
HI THERE
```

Given some rules that specify a function, how do we write these rules as a LOGO procedure? To illustrate, let us write LOGO procedures for the two pair lists, /NEXT/ and /REVERSE/, described in Section 2.4. The rules given for defining these functions were difficult to express in pair list notation. They are, however, easily expressed in the form of LOGO procedures.

```
TO NEXT /W/
1 OUTPUT WORD OF (BUTFIRST OF /W/) AND (FIRST OF /W/)
END
```

The procedure NEXT simply moves the first character of its input word /W/ to the end of the word, thereby accomplishing the desired effect.

```
←PRINT NEXT OF "ABCD"
BCDA
←PRINT NEXT OF "CAT"
ATC
```

The other function, REVERSE, can be written as a recursive procedure, corresponding to the following rule: the reverse of the one-letter word is the word itself; the reverse of a larger

word is the word formed by putting the last letter of the word in front of the reverse of the rest of the word. This rule carries directly over to the procedure.

```
TO REVERSE /W/
1 TEST IS COUNT OF /W/ "1"
2 IF TRUE OUTPUT /W/
3 OUTPUT WORD OF (LAST OF /W/) AND
   REVERSE OF (BUTLAST OF /W/)
END

←PRINT REVERSE OF "CAT"
TAC
←PRINT REVERSE OF NEXT OF "CAT"
CTA
←PRINT REVERSE OF REVERSE OF "CAT"
CAT
```

REVERSE's domain includes English words and other non-numbers as well as numbers. It is interesting to note that there is a well-known function on integers whose LOGO procedure is identical in form to the procedure for REVERSE.

```
TO FACTORIAL /N/
1 TEST IS /N/ "Ø"
2 IF TRUE OUTPUT "1"
3 OUTPUT PRODUCT OF /N/ AND
   FACTORIAL OF (DIFFERENCE OF /N/ AND 1)
END

←PRINT FACTORIAL OF 4
24
←PRINT REVERSE OF FACTORIAL OF 4
42
```

The factorial definition is usually expressed in functional form as:

FACTORIAL (Ø) = 1
FACTORIAL (N) = N * FACTORIAL (N-1)

and this is the computation used in the LOGO procedure.

Given a function specified by appropriate rules, the problem of representing the function as a LOGO procedure is straightforward, in principle at least. And most of our work seems to proceed in this direction. But there are valuable pedagogical reasons for going in the other direction too. For example, imitating "nature", which realizes processes without making their operation transparent to us, we can study the following kind of problem. Given a LOGO procedure, does it represent a function and, if so, what function? We begin by considering the relatively "blind" case where we are given a LOGO procedure to use but are not allowed to see the instructions comprising its definition.

3.2 Black Boxes

By a *black box* we mean a LOGO procedure whose instructions are not open to our inspection. Assume we are given the name of such a procedure, say BLACKBOX, and are told that it takes one input. Usually we can list a LOGO procedure. Thus, to list the procedure SECOND, we simply type LIST SECOND. LOGO responds,

```
TO SECOND /ANYTHING/
1 OUTPUT FIRST OF BUTFIRST OF /ANYTHING/
END
```

that is, LOGO prints out the title and body of the procedure. Perhaps we will then try to list the procedure to "see" what it does. But this is not allowed with the procedure BLACKBOX, since someone has previously executed the LOGO command BURY BLACKBOX, which renders it opaque until it has been appropriately "unburied". So, if we try to list BLACKBOX, this is what happens:

```
←LIST BLACKBOX
TO BLACKBOX /NUMBER/
END
←
```

How can we try to find out whether BLACKBOX is, in fact, a function? How can we try to determine its operation -- that is, its domain and range elements and the rules describing how these elements are associated? We might begin simply by trying to use BLACKBOX with various inputs, as follows.

```
←PRINT BLACKBOX OF Ø
Ø
←PRINT BLACKBOX OF 1Ø
2Ø
←PRINT BLACKBOX OF "A"
A IS NOT IN MY DOMAIN
←PRINT BLACKBOX OF 7
14
←PRINT BLACKBOX OF 1ØØ
2ØØ
```

Based on these results, we might conclude that BLACKBOX OF /N/ is a function which doubles the number /N/. Of course, we haven't tried very many inputs. So, to test our guess we could try BLACKBOX with a large number of cases, say two hundred. This would entail a large amount of tedious work, we might as well write a procedure TRY to do this for us.

```
TO TRY /N/                          (/N/ is the current input for
1Ø TEST IS /N/ 2ØØ                  testing with BLACKBOX)
2Ø IF TRUE STOP
3Ø PRINT SENTENCE OF /N/            (Print input and result)
   AND (BLACKBOX OF /N/)
4Ø TRY (SUM OF /N/ AND 1)           (Repeat with input increased by 1)
END
```

Let's start the procedure:

```
←TRY -1
-1 -1
Ø Ø
1 2
2 4
3 6
4 8
5 1Ø
: : :      (and so on, until,)
: : :
199 398    (Finally!!)
←
```

If we had anticipated the large stack of uninteresting paper that would result, we really would have written a better TRY procedure. This new version of TRY only prints interesting information -- it prints results only when the test fails.

```
TO TRY /N/
1Ø TEST IS /N/ 1ØØØ                    (We have upped the limit)
2Ø IF TRUE STOP
3Ø TEST IS (BLACKBOX OF /N/)
    (DOUBLE OF /N/)
4Ø IF FALSE PRINT SENTENCE OF /N/
    AND (BLACKBOX OF /N/)
5Ø TRY (SUM OF /N/ AND 1)
END
```

This version of TRY compares the outputs of BLACKBOX and DOUBLE for each successive /N/ (in Line 30). It prints the input and the associated output of BLACKBOX only when the output is not the same as for DOUBLE. Let's try TRY.

```
←TRY -1ØØØ
←
```

Sometime later (after 2000 uses of BLACKBOX), TRY terminated without any intermediate printout. Can we conclude that BLACKBOX is identical to DOUBLE? Of course not -- perhaps BLACKBOX is really the following function

```
TO BLACKBOX /N/
1 TEST NUMBERP /N/
2 IF FALSE OUTPUT SENTENCE OF
  /N/ AND "IS NOT IN MY DOMAIN"
3 OUTPUT DOUBLE OF (FIRST3 OF /N/)
END
```

where FIRST3 OF /N/ is a procedure which outputs the first three digits of its input. Thus,

```
←PRINT FIRST3 OF "9999"
999
←PRINT FIRST3 OF 222
222
←PRINT FIRST3 OF 35
35
```

This BLACKBOX has the same output as DOUBLE for inputs less than 1ØØØ. But for larger inputs, it doubles only the first three digits of the number.

```
←PRINT BLACKBOX OF 1ØØØ
2ØØ
←PRINT BLACKBOX OF 9999999999
1998
```

We would have found that our conjecture (BLACKBOX is equivalent to DOUBLE) did not hold if the limit in TRY were 1ØØ1 rather than 1ØØØ. Alternatively, we could have written a TRY procedure which runs indefinitely, stopping only if it encounters a failure. We can write such a procedure along the following lines.

```
TO TRY /N/
1Ø TEST IS (BLACKBOX OF /N/) (GUESSFUNCTION OF /N/)
2Ø IF TRUE TRY (SUM OF /N/ AND 1)
3Ø IF FALSE PRINT SENTENCE OF /N/ AND (BLACKBOX OF /N/)
END
```

Whenever GUESSFUNCTION (so far this has been DOUBLE in our example) fails to match BLACKBOX, TRY prints the input that fails along with the associated BLACKBOX output. After this occurs with DOUBLE, we might probe BLACKBOX further, possibly using some strategy other than systematic search. This might lead to a better conjecture. If it did, we could replace DOUBLE with an improved GUESSFUNCTION procedure corresponding to this improved conjecture and repeat the TRY procedure.

Of course, this kind of strategy is not guaranteed to work. But it can provide a good model for students for investigating many kinds of functions and non-functions. As part of this study, students can invent their own black boxes and try these out with other students. Procedures such as the following can be used

with a new black box, first letting a user interrogate the black box (by giving him black box outputs for requested inputs) and then (when the user declares that he is ready to guess the function) for testing whether he can give the output corresponding to a series of random or preselected inputs.

```
TO BOXTEST
1 PRINT "TRY TO FIGURE OUT THE RULE IN MY SECRET BOX.  TYPE
  IN ANY INPUT YOU LIKE AND I'LL TYPE BACK THE BOX'S OUTPUT.
  WHEN YOU THINK YOU KNOW HOW THE BOX WORKS, TELL ME BY
  TYPING FIN."
2 RESPOND
3 PRINT "NOW I'LL GIVE YOU SOME INPUTS AND YOU TELL ME WHAT
  YOU THINK THE BOX WOULD OUTPUT FOR THESE."
4 CHALLENGE
END
```

The outer procedure BOXTEST merely introduces and then calls, in turn, the two subprocedures RESPOND and CHALLENGE. These can be written in many ways, corresponding to a variety of different test conditions. The following instance of RESPOND permits the user to interrogate the black box with as many inputs as he wishes. (The black box procedure is again called BLACKBOX.)

```
TO RESPOND
1 TYPE "INPUT:   "              (Label input)
2 MAKE
    NAME:  "INPUT"              (Wait for user to type input
    THING:  REQUEST             to black box)
3 TEST IS /INPUT/ "FIN"         (When user is done
4 IF TRUE STOP                  go back to BOXTEST)
5 PRINT SENTENCE OF             (Label output)
   "OUTPUT:  "                  (Compute and
   BLACKBOX OF /INPUT/          print output)
6 RESPOND                       (Repeat process)
END
```

For example, if BLACKBOX is REVERSE as defined above, RESPOND would work as follows, where the user's type-in is underscored to distinguish it from LOGO's printout.

```
←RESPOND
INPUT:    Ø
OUTPUT:   Ø
INPUT:    99
OUTPUT:   99
INPUT:    123456789
OUTPUT:   987654321
INPUT:    FIN
←
```

The following version of CHALLENGE presents the student with successive test inputs from a predetermined list, /TEST INPUTS/. The student types in what he thinks BLACKBOX will output for each given input. If he is wrong, he is given the correct output. The process stops either when he correctly answers /NRIGHT/ questions in succession with no intervening mistakes, or when the list /TEST INPUTS/ is exhausted.

CHALLENGE initializes the list /SCORES/ on which the student's right and wrong scores will be recorded (as "T" or "F", respectively). It then calls a subprocedure QUIZ.

```
TO CHALLENGE
1 MAKE
   NAME:  "SCORES"
   THING:  Ø
2  QUIZ /TEST INPUTS/
END
```

The procedure QUIZ presents the questions and scores the student's answers. We use "SCORES" to label the number of consecutive correct answers. We increment /SCORES/ by one in the case of a right answer and set it back to Ø for a wrong answer.

```
TO QUIZ /TEST INPUTS/
1 TEST IS /SCORES/                      (Is there a run of /NRIGHT/
  /NRIGHT/                              correct answers?  /NRIGHT/
2 IF TRUE PRINT "GOOD! IT APPEARS       must be specified previously)
  YOU KNOW WHAT'S IN THE BOX"
3 IF TRUE STOP
4 TEST IS /TEST INPUTS/ /EMPTY/         (If the questions are exhausted)
5 IF TRUE PRINT "YOU DON'T              (The student has failed to
  KNOW WHAT'S IN THE BOX.               guess the function)
  SEE ME TO FIND OUT OR TO GET
  ANOTHER CHANCE WITH SOME NEW
  TEST INPUTS"
6 IF TRUE STOP                          (So, stop)
7 TYPE SENTENCE OF "INPUT:"             (Print next question)
  AND (FIRST OF /TEST INPUTS/)
8 TYPE "  OUTPUT: "                     (Label student's answer)
9 MAKE
    NAME:  "ANS"                        (Wait for student's answer)
    THING:  REQUEST
1Ø MAKE
    NAME:  "RIGHT ANS"                  (Compute the correct answer)
    THING:  BLACKBOX OF (FIRST
            OF /TEST INPUTS/)
11 TEST IS /ANS/ /RIGHT ANS/            (Is the student's answer correct?)
12 IF TRUE MAKE "SCORES"                (Add 1 to /SCORES/)
   (SUM /SCORES/ AND 1)
13 IF TRUE PRINT "RIGHT!"
14 IF FALSE MAKE "SCORES" Ø             (Set /SCORES/ back to Ø)
15 IF FALSE PRINT SENTENCE OF           (Print correct answer
   "WRONG! THE RIGHT ANSWER IS"
   AND /RIGHT ANS/
16 QUIZ (BUTFIRST OF /TEST INPUTS/)     (Continue quiz with next
END                                     question)
```

The effects of CHALLENGE and QUIZ are illustrated next in the context of a complete run of BOXTEST. Again, all user typing is underscored in this hypothetical transcript.

```
BOXTEST
TRY TO FIGURE OUT THE RULE IN MY SECRET BOX.  TYPE IN ANY
INPUT YOU LIKE AND I'LL TYPE BACK THE BOX'S OUTPUT.  WHEN
YOU THINK YOU KNOW HOW THE BOX WORKS, TELL ME BY TYPING FIN.
INPUT:  Ø
OUTPUT: Ø
INPUT:  1Ø
OUTPUT: 1
INPUT:  234
OUTPUT: 9
```

```
INPUT:  1234
OUTPUT: Ø
INPUT:  77
OUTPUT: 4
INPUT:  543
OUTPUT: 2
INPUT:  551
OUTPUT: 1
INPUT: FIN
NOW I'LL GIVE YOU SOME INPUTS AND YOU TELL ME WHAT YOU
THINK THE BOX WOULD OUTPUT FOR THESE
INPUT: 1ØØ  OUTPUT: 1
RIGHT!
INPUT: 72  OUTPUT: 9
RIGHT!
INPUT: 7531357  OUTPUT: 2
WRONG!  THE RIGHT ANSWER IS 1
INPUT: 22222  OUTPUT: Ø
RIGHT!
INPUT: 111222333  OUTPUT: 8
RIGHT!
INPUT: 314159  OUTPUT: 3
RIGHT!
INPUT: 1Ø2Ø1Ø3Ø7  OUTPUT: 4
RIGHT!
GOOD!  IT APPEARS YOU KNOW WHAT'S IN THE BOX
←
```

In the example, /NRIGHT/ was "4" and /TEST INPUTS/ was "100 72 7531357 22222 111222333 314159 102010307 52". The procedure BLACKBOX was

```
TO BLACKBOX /N/
1 TEST NUMBERP OF /N/
2 IF FALSE OUTPUT SENTENCE OF /N/
    "IS NOT IN MY DOMAIN"
3 OUTPUT LAST OF DIGITSUM OF /N/
END


TO DIGITSUM /N/
1 TEST IS /N/ /EMPTY/
2 IF TRUE OUTPUT Ø
3 OUTPUT SUM OF (FIRST OF /N/) AND
  (DIGITSUM OF BUTFIRST OF /N/)
END
```

To paraphrase, DIGITSUM OF /N/ outputs the sum of the individual digits of the integer /N/ and BLACKBOX outputs the last digit of the result.

Students can modify BOXTEST in a number of ways. For example, some might limit the number of probes a user is permitted during the exploratory phase (in RESPONSE) or assign time limits for answering questions during the quiz phase (in CHALLENGE). Inventive students can devise a number of different kinds of black boxes, for non-functions as well as functions, including non-numerical functions such as letter substitution ciphers and permutation codes. Also, it is amusing to try tricks such as random operations or time-dependent operations, in defining some of their black boxes.

In later projects, after working on a series of different kinds of black box programs, students write procedures for *random* generation of black box functions of variable as well as fixed form. To begin with, let's see how we can write a LOGO procedure for generating a function (that is, for writing another LOGO procedure!).

3.3 Automatic Composition and Generation of Functions

We start with a procedure for repeating any LOGO sentence /S/, which could be a function name, a specified number of times.

```
TO REPEAT /S/ /NTIMES/
1Ø TEST ZEROP OF /NTIMES/          (When /NTIMES/ is zero)
2Ø IF TRUE OUTPUT /EMPTY/          (add on the empty word and stop)
3Ø OUTPUT SENTENCE OF /S/ AND
  (REPEAT OF /S/ DIFFERENCE        (Otherwise, add on an /S/ and
  OF /NTIMES/ AND 1)               repeat the process one less time)
END
```

The procedure is a simple recursion. Its effect is clear.

```
←PRINT REPEAT OF "HA" 4
HA HA HA HA
←PRINT REPEAT OF "FOO OF" 3
FOO OF FOO OF FOO OF
```

We can incorporate REPEAT in procedures for chaining one or more functions a specified number of times, and then executing the resulting composition. Let's show this first with a particular function, say DOUBLE.

```
TO REDOUBLE /NUMBER/ /NTIMES/
1Ø MAKE
    NAME:  "NDOUBLES"
    THING:  REPEAT OF "DOUBLE OF" AND /NTIMES/
2Ø MAKE
    NAME:  "COMPOSED FUNCTION"
    THING:  SENTENCE OF SENTENCE OF
            "OUTPUT" AND /NDOUBLES/ AND /NUMBER/
3Ø PRINT /COMPOSED FUNCTION/
4Ø DO /COMPOSED FUNCTION/
END
```

After making a sentence /NDOUBLES/, out of repeated "DOUBLE OF" terms, REDOUBLE puts the command "OUTPUT" in front and an input, /NUMBER/, at the end. Line 3Ø prints the resulting sentence. Line 4Ø uses the DO command to execute the sentence as a LOGO command. REDOUBLE works as follows.

```
←PRINT REDOUBLE OF 3 4
OUTPUT DOUBLE OF DOUBLE OF DOUBLE OF DOUBLE OF 3
48

←PRINT REDOUBLE 13579 5
OUTPUT DOUBLE OF DOUBLE OF DOUBLE OF DOUBLE OF DOUBLE OF 13579
434528
```

Let's extend this idea to include composition of other functions as well as DOUBLE. Assuming we have previously defined a LOGO

procedure /FUN/ for some function which takes one input, we can write a procedure to compose and execute /FUN/ OF /INPUT/, as follows.

```
TO COMPOSE /FUN/ /INPUT/ /NTIMES/
1Ø MAKE "NFUNS" REPEAT OF /FUN/ /NTIMES/
2Ø MAKE "CHAIN" SENTENCE OF SENTENCE OF
   "OUTPUT" AND /NFUNS/ AND /INPUT/
3Ø PRINT /CHAIN/
4Ø DO /CHAIN/
END
```

To illustrate COMPOSE, assume the function we want to compose is the following.

```
TO DUB /NUMBER/
1 OUTPUT WORD OF /NUMBER/ AND /NUMBER/
END

←PRINT DUB OF 12345
1234512345
```

Now, let's illustrate COMPOSE with DUB.

```
←PRINT COMPOSE OF "DUB" 1 4
OUTPUT DUB DUB DUB DUB 1
1111111111111111
```

/FUN/ can itself be a composed function. Thus,

```
←PRINT COMPOSE OF "DUB OF DOUBLE OF" 1 3
OUTPUT DUB OF DOUBLE OF DUB OF DOUBLE OF DUB OF DOUBLE OF 1
88888888
←PRINT COMPOSE OF "DOUBLE DUB" 2 3
OUTPUT DOUBLE DUB DOUBLE DUB DOUBLE DUB 2
177777776
←
```

The next step in our plan for randomly generating blackbox functions, using a procedure like COMPOSE, is to chain together

a random number of functions, randomly selected from some prescribed set. Thus, we'll need a procedure CHOOSE for choosing a function, and a set of functions over which to make this choice.

Let's choose a set of ten functions and list their names in the sentence /FUNSET/. For example, let's begin with the choice

```
MAKE
   NAME:  "FUNSET"
   THING:  "REV DUB BF BL RUN LAG FF LF FL LL"
```

where the ten functions named are described as follows.

REV is the previously defined REVERSE.

```
←PRINT REV OF "CAT"
TAC
←
```

DUB is the procedure defined just above.

```
←PRINT DUB OF "CAT"
CATCAT
←
```

BF and BL are the built-in LOGO operations BUTFIRST and BUTLAST.

```
←PRINT BUTFIRST OF "CAT"
AT
←PRINT BUTLAST OF "CAT"
CA
←
```

RUN and LAG are right and left rotational shifts.

```
←PRINT RUN OF "CATS"
SCAT
←PRINT LAG OF "CATS"
ATSC
←
```

FF, FL, LF, and LL extend their input on both sides by the indicated first and last characters.

```
←PRINT FF OF "CAT"
CCATC
←PRINT FL OF "CAT"
CCATT
←PRINT LF OF "CAT"
TCATC
←PRINT LL OF "CAT"
TCATT
←
```

CHOOSE selects one of these ten functions using an input /R/ as follows.

```
TO CHOOSE /FUNSET/ /R/
1Ø TEST IS /R/ Ø
2Ø IF TRUE OUTPUT FIRST OF /FUNSET/
3Ø OUTPUT CHOOSE (BUTFIRST OF /FUNSET/)
   (DIFFERENCE OF /R/ AND 1)
END
```

```
←PRINT CHOOSE OF /FUNSET/ Ø
REV
←PRINT CHOOSE OF /FUNSET/ 9
LL
←
```

We can use RANDOM as the number input to CHOOSE to pick out one of the ten functions at random. Then, using CHOOSE, we can write a procedure for composing /N/ randomly chosen functions.

```
TO RANDOMCHAIN /N/
1 TEST ZEROP /N/
2 IF TRUE OUTPUT /EMPTY/
3 OUTPUT SENTENCE OF
   CHOOSE OF /FUNSET/ AND RANDOM
   RANDOMCHAIN OF (DIFFERENCE OF /N/ AND 1)
END
```

Each of the /N/ times RANDOMCHAIN is called, it appends another randomly chosen function from /FUNSET/ to its output sentence. When /N/ is itself a random number, the resulting compositions are variable in both size and composition. For example,

```
←PRINT RANDOMCHAIN OF RANDOM
RUN BF REV BF FF
←PRINT RANDOMCHAIN OF RANDOM
DUB LAF LF FL REV DUB DUB
←PRINT RANDOMCHAIN OF RANDOM
BF BL LL
←
```

Now we can write a random function generation procedure which uses RANDOMCHAIN and names the resulting composed function "FUNCTION".

```
TO GENERATE
1Ø MAKE
    NAME:  "SIZE"
    THING:  SUM OF RANDOM AND 2
2Ø MAKE
    NAME:  "FUNCTION"
    THING:  RANDOMCHAIN OF /SIZE/
END
```

The number of composed elements, /SIZE/, is computed as a random number between 2 and 11 (in Line 1Ø). Thus, the limiting cases of composing no functions or just a single function are excluded. Finally, given an input, we can use the randomly generated function, /FUNCTION/, in our BLACKBOX.

```
TO BLACKBOX /INPUT/
1Ø DO SENTENCE OF SENTENCE OF
   "OUTPUT" AND /FUNCTION/ AND /INPUT/
END
```

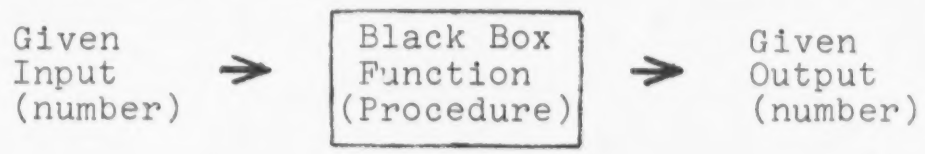
To make a new BLACKBOX then, all we have to do is invoke the procedure GENERATE:

```
←GENERATE
←
```

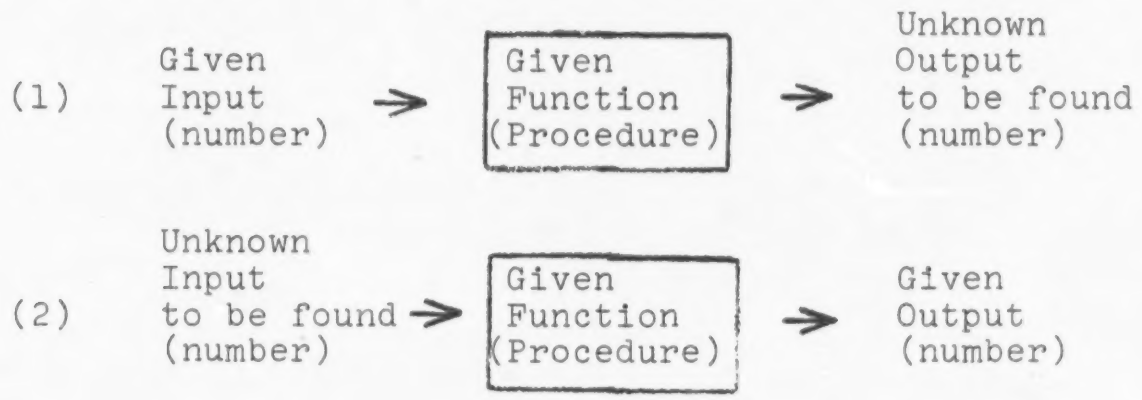
Assume that the resulting /FUNCTION/ is REV LAG DUB. Then BLACKBOX will evaluate that function for prescribed inputs.

```
←PRINT BLACKBOX OF 2468Ø
2Ø8642Ø864
←PRINT BLACKBOX OF 1Ø
1Ø1Ø
←PRINT BLACKBOX OF 1
11
←PRINT BLACKBOX OF 123
132132
```

Generating black boxes of this kind is a great deal easier than solving them. Perhaps that's because the boxes are relatively complex objects - procedures - instead of just numbers, like a box's input and output. We can make a schematic diagram of the situation we have been studying, to clarify this. Here is a picture of our problem:



Two variants of this picture are evident at once.



The two new situations are: (1) given an input and a function, determine the output; and (2) given a function and an output, determine the input. The first situation does not ordinarily pose any difficulty, since the output of a function is uniquely determined, given an input and the procedure which specifies the function. The second situation, however, is of a different sort. Let's consider it further.

3.4 Inverse Functions

Given a procedure representing a function, and an output of the function, we can write guessing or searching procedures for finding the associated input or inputs. Even when we know the domain of allowable inputs, random guessing or systematic search methods will be extremely inefficient if the domain is not finite.

Consider, for example, the blackbox functions we just generated in Section 3.3, such as REV OF LAG OF DUB. We can easily write procedures such as the following for use with these.

```
TO SET-UP /INPUT/ /FUNCTION/
1 MAKE
    NAME:  "OUTPUT"
    THING:  DO SENTENCE OF /FUNCTION/ AND /INPUT/
END
```

This SET-UP procedure simply computes the output, /OUTPUT/, of the given function /FUNCTION/ applied to the input /INPUT/. The procedure is performed in advance of the following one, which is used for guessing an input corresponding to a given function and one of its outputs.

```
TO GUESS-INPUT /FUNCTION/ /OUTPUT/
1Ø TYPE "GUESS THE INPUT:  "
2Ø MAKE
    NAME:  "GUESS"                          (This is where the user
    THING:  REQUEST                         makes his guess)
3Ø MAKE
    NAME:  "RESULT"
    THING:  DO SENTENCE OF /FUNCTION/ AND /GUESS/
4Ø TEST IS /RESULT/ /OUTPUT/
5Ø IF FALSE PRINT SENTENCE OF SENTENCE OF
   "WRONG.  YOUR GUESS GIVES THE OUTPUT  "
   /GUESS/ "  GUESS AGAIN."
6Ø IF FALSE GUESS-INPUT /FUNCTION/ /OUTPUT/
7Ø PRINT "GOOD!  YOU'VE GOT IT."
END
```

This is how GUESS-INPUT might be used. The guesser is told to call the procedure with the inputs "BLACKBOX" and 2086420864. He might start by listing the function named "BLACKBOX", as follows.

```
←LIST BLACKBOX
TO BLACKBOX /NUMBER/
1 OUTPUT REV OF LAG OF DUB OF /NUMBER/
END
←
```

Since he knows the operation of the procedures REV, LAG, and DUB, he does not need to list these. Instead, he begins his guessing, as follows. (His typing is underscored.)

```
←GUESS-INPUT "BLACKBOX" 2Ø86420864
GUESS THE INPUT:  8642Ø
WRONG.  YOUR GUESS GIVES THE OUTPUT  8Ø2468Ø246  GUESS AGAIN.
GUESS THE INPUT:  2468Ø
GOOD!  YOU'VE GOT IT.
←
```

Quite clearly, an insightful user will do better than either the wild guesser or the plodder. For instance, with the guessing problem used in our example, a systematic search procedure would require 2468Ø trials before it was successful. A random guessing procedure, even if it were improved so as to exclude previously rejected trials, would require over a billion trials on the average, even if we knew the input was bounded by the output.

Rather than consider whether we can significantly improve this situation by incorporating more efficient searching strategies, let's consider the more directed kind of "guessing" that was employed in the REV LAG DUB example.

Instead of the composite function, let's consider the three separate functions in the chain DUB, LAG, and REV. In GUESS-INPUT, if /FUNCTION/ is chosen as "DUB" and /OUTPUT/ as 2468Ø2468Ø, we do not need to think a great deal before responding with a "guess" of 2468Ø as the associated input. Similarly, if /FUNCTION/ is "LAG" and /OUTPUT/ is 468Ø2468Ø2, we readily assert that the correct input is 2468Ø2468Ø. Finally, if /FUNCTION/ is "REV" and /OUTPUT/ is 2Ø8642Ø864, we are confident that the input is 468Ø2468Ø2. Moreover, using these functions, or any of those we have encountered thus far, we would be able to do the same with virtually any choice of number for /OUTPUT/.

How do we do this so easily? What kind of procedure do we employ on /FUNCTION/ and /OUTPUT/ to compute /INPUT/? Can we use *that* procedure instead of the others we have been writing in Line 2Ø of GUESS-INPUT?

In all these cases what we do is pretty much the same. And a small amount of introspection suffices to reveal the underlying mechanism. Let's make it explicit. Given a function F and an output /O/, to determine the input we evaluate F's *inverse function*, with /O/ as an input. The inverse function is a procedure for undoing the effect of the given function. In particular, the inverse functions for DUB, LAG, and REVERSE -- call them UNDUB, UNLAG, and UNREVERSE are as follows.

```
TO UNDUB /W/
1Ø MAKE "C" QUOTIENT OF (COUNT OF /W/) AND 2
2Ø MAKE "FIRSTHALF" FIRSTN OF /C/
3Ø MAKE "LASTHALF" LASTN OF /C/
4Ø TEST IS /FIRSTHALF/ /LASTHALF/
5Ø IF TRUE OUTPUT /FIRSTHALF/
6Ø OUTPUT /EMPTY/
END
```

Thus, UNDUB splits the word in half two ways (using the procedures FIRSTN OF /C/ and LASTN OF /C/ which get the first /C/ and last /C/ characters of a word). If the halves are identical, it outputs one of them; otherwise, it outputs the empty word.

```
←PRINT UNDUB OF "12345671234567"
1234567
←PRINT UNDUB OF "12345678"
←                                (It outputs the empty word here)
```

UNLAG is already familiar to us.

```
TO UNLAG /W/
1 OUTPUT RUN OF /W/
END
```

That is, to undo LAG one simply does the procedure RUN (and vice versa).

```
←PRINT UNLAG OF "12345"
51234
```

The case of UNREV is even simpler.

```
TO UNREV /W/
1 OUTPUT REV OF /W/
END
```

That is, the inverse of REV is just REV itself.

```
←PRINT UNREV OF "12345"
54321
```

The following examples show the effect of composing a function with its inverse in either order.

```
←PRINT DUB OF UNDUB OF "ABCABC"
ABCABC
←PRINT UNDUB OF DUB OF "ABCDEF"
ABCDEF
```

DUB OF UNDUB and UNDUB OF DUB leave an input unchanged. They are identical in effect to the procedure SAME:

```
TO SAME /INPUT/
1 OUTPUT /INPUT/
END
```

This result also holds for the pairs LAG and UNLAG and REV and UNREV.

In general, if a function FUN has an inverse UNFUN, then for any input /I/ in the domain of FUN OF /I/, UNFUN OF FUN OF /I/ and SAME OF /I/ express the same thing though they are very different procedures. Sometimes the inverse function of the inverse function is the original function, that is, FUN is also the inverse of UNFUN so FUN OF UNFUN OF /I/ is also equal to SAME OF /I/. This duality is important in applications like encoding and decoding where we want both DECODE OF ENCODE and ENCODE OF DECODE to preserve their inputs.

An example of a case where the inverse exists in only one direction is the function DOUBLE, defined over the domain of the integers as follows.

```
TO DOUBLE /NUMBER/
1 OUTPUT PRODUCT OF /NUMBER/ AND 2
END

←PRINT DOUBLE OF 7
14
←
```

DOUBLE has a one-sided inverse UNDOUBLE:

```
TO UNDOUBLE /NUMBER/
1 OUTPUT QUOTIENT OF /NUMBER/ AND 2
END

←PRINT UNDOUBLE OF 14
7
←
```

UNDOUBLE, like DOUBLE, is defined over all integers and always outputs an integer. The LOGO operation QUOTIENT outputs the integer result of a division, ignoring any remainder. So UNDOUBLE is not always the same operation as exact division by 2. For example,

```
←PRINT UNDOUBLE OF 15
7
←
```

Because of this, DOUBLE OF UNDOUBLE fails to preserve odd integers. For example,

```
←PRINT UNDOUBLE OF DOUBLE OF 15
15
```

going the other way, however,

```
←PRINT DOUBLE OF UNDOUBLE OF 15
14
```

Thus, if we ask what input to DOUBLE causes 15 to be output, UNDOUBLE gives the wrong answer. But that is easy to fix. The real problem is that not all allowable inputs to UNDOUBLE are possible outputs of DOUBLE. For UNDOUBLE to be a right-side inverse to DOUBLE it would need to have a domain exactly equal to the possible outputs of DOUBLE.

Sometimes we have the opposite problem. Instead of no inputs that give the desired output, there are more than one. For example, consider a built-in LOGO function like BUTLAST. What input to BUTLAST causes "BO" to be output? The input may have been "BOA", "BOB", ..., "BOY", "BOZ", ... "BO9", "BO?" or even "BO ANYWORD". An arithmetic example of the same kind is the following procedure for squaring an integer.

```
TO SQUARE /N/
1 OUTPUT PRODUCT OF /N/ AND /N/
END
```

With this function we get both problems. Only certain integers are possible outputs for SQUARE and for each of these (except one) there are two possible inputs.

The blackbox problem for the functions BUTLAST and SQUARE (finding the input for which the function gives a specified output /OPT/) does not have a general solution. We therefore focus our attention on functions such as REV, LAG, and DUB, which have inverses. Are there procedures for finding an inverse function when there is one? The problem which initiated our inquiry was to find an inverse for a composition of functions (REV OF LAG OF DUB) when we know the inverses of each of the constituents. To gain insight into this problem we shall look at the black box problem once more, this time in the context of equations.

4. Equations

The two black box problems for functions -- finding an unknown function given input-output pairs, and finding an unknown input given the function and an output -- have their analogs in black box problems for *equations*. By an equation is meant a statement of equality between two expressions (such as the standard high school algebra expressions involving arithmetic operations on numbers). Thus, 3 x + 5 = 26 is a numerical or algebraic equation in the usual notation, and WORD OF "SEC" AND /SECOND/ = "SECRET" is a LOGO equation.

The problem of finding the unknown things -- x and /SECOND/ in the examples -- is called "solving equations". This is related to the problem of "evaluating expressions". (For example, given the number x, say x = 7, finding the number expressed by 3 x + 5;

or, given the word /SECOND/, say /SECOND/ = "RET", finding the word expressed by WORD OF "SEC" AND /SECOND/.) We can write procedures to illuminate both of these problems.

4.1 Evaluating and Testing Expressions

To evaluate a LOGO expression using LOGO, we simply need to command LOGO to perform it. We have been doing this all along, of course. Thus, to evaluate the expression WORD OF /W/ AND /W/ we merely type it (preceded by a PRINT if we want to see the result of evaluation).

```
←PRINT WORD OF /W/ AND /W/
CANCAN                                  (if /W/ is "CAN")
```

LOGO can also evaluate *arithmetic* expressions written in this same "prefix" (or functional) notation. For example:

```
←PRINT SUM OF 2 AND PRODUCT OF 3 AND DIFFERENCE OF 4 AND 5
-1
```

In the more familiar ("infix") notation the same expression is written:

```
2 + 3 * (4 - 5)
```

LOGO cannot directly evaluate arithmetic expressions written in infix notation. But we can write programs to translate expressions written in infix to prefix form, and thereby enable LOGO to evaluate them. A *general* translation program of this kind, PREFIX, is developed and discussed in the unit on algebra story problems. In this unit we shall write more limited programs to deal with special cases, i.e., expressions having known forms.

Consider infix algebraic expressions of the form /TIMESNUM/ /BOX/ + /SUMNUM/, where the multiplication operation is implicit, /TIMESNUM/ and /SUMNUM/ are given numbers and /BOX/ is a number not yet specified -- let's call it the missing number. An example is 11 x + 13. This expression can be written in LOGO prefix notation as SUM OF (PRODUCT OF 11 AND /X/) AND 13. After /X/ is given, LOGO can evaluate the expression. It is not difficult for beginning students of algebra to write procedures such as the following, to perform such evaluations.

```
TO EVALUATE /EXPRESSION/
1Ø MAKE
    NAME:  "TIMESNUM"
    THING:  FIRST OF /EXPRESSION/
2Ø MAKE
    NAME:  "SUMNUM"
    THING:  LAST OF /EXPRESSION/
3Ø MAKE
    NAME:  "BOXNAME"          (Get user's name for the missing number)
    THING:  SECOND OF /EXPRESSION/
4Ø PRINT SENTENCE OF SENTENCE OF
   "FOR WHAT" /BOXNAME/ "?"
45 TYPE SENTENCE OF /BOXNAME/ "="
5Ø MAKE
    NAME:  "BOX"                  (User types in the missing number)
    THING:  REQUEST
6Ø MAKE                           (Evaluate expression)
    NAME:  "RESULT"
    THING:  SUM OF (PRODUCT OF
            /TIMESNUM/ /BOX/) AND /SUMNUM/
7Ø PRINT SENTENCE OF SENTENCE OF
   /EXPRESSION/ "=" /RESULT/
END
```

Here is an example of the use of EVALUATE. (The user's typing is underscored.)

```
←EVALUATE "17 N + 42"
FOR WHAT N?
N = 3
17 N + 42 = 93
←
```

Instead of making up our own expressions, we can write a procedure for generating random expressions of this form, for example:

```
TO RANDOMEXP
1Ø OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
   RANDOMNUMBER OF 999
   RANDOMWORD OF 3
   "+"
   RANDOMNUMBER OF 999
END
```

The subprocedures RANDOMNUMBER /N/ and RANDOMWORD /N/ generate a random number in the range Ø through /N/, and a random word whose size is in the range 1 through /N/.

```
←PRINT RANDOMEXP
71 GB + 254
←PRINT RANDOMEXP
2Ø3 ZAZ + 1
←
```

Using RANDOMEXP we can write quiz programs to test someone else's skill in evaluating expressions. First, let's modify the evaluation procedure so it is given /BOX/ as an input instead of as a typed-in response to a REQUEST.

```
TO EVAL /EXPRESSION/ /BOX/
1Ø OUTPUT SUM OF (PRODUCT OF FIRST OF /EXPRESSION/
   AND /BOX/) AND LAST OF /EXPRESSION/
END

←PRINT EVAL OF "32 T + 5" 1Ø
325
```

First we'll write a program that administers drill problems.

```
TO EVAL-DRILL
1Ø MAKE
    NAME:  "EXP"
    THING:  RANDOMEXP
2Ø MAKE
    NAME:  "BOX"
    THING:  RANDOM
3Ø PRINT SENTENCE SENTENCE SENTENCE SENTENCE SENTENCE SENTENCE
   "WHAT IS THE RESULT OF EVALUATING"
   /EXP/
   "FOR"
   SECOND OF /EXP/
   "="
   /BOX/
   "?"
4Ø MAKE
    NAME:  "ANSWER"
    THING:  REQUEST
5Ø TEST IS /ANSWER/ (EVAL OF /EXP/ AND /BOX/)
6Ø IF TRUE PRINT "RIGHT!"
7Ø IF FALSE PRINT SENTENCE OF
   "WRONG.  THE ANSWER IS"
   EVAL OF /EXP/ AND /BOX/
8Ø EVAL-DRILL
END
```

Here's what EVAL-DRILL does.

```
*EVAL-DRILL
WHAT IS THE RESULT OF EVALUATING 37 PY + 12 FOR PY = 4?
*17Ø
WRONG.  THE ANSWER IS 16Ø
WHAT IS THE RESULT OF EVALUATING 19 VV + 91 FOR VV = 9?
*262
 : : :
```

Let's make a slight modification of EVAL-DRILL and change the user's activity from drill to something that may appear very different -- checking or grading. Instead of giving answers to problems, the user's task is deciding whether or not proposed answers to problems are actually correct.

```
TO EVAL-CHECK
1Ø MAKE
    NAME:  "EXP"
    THING:  RANDOMEXP
2Ø MAKE
    NAME:  "BOX"
    THING:  RANDOM
3Ø MAKE
    NAME:  "RESULT"
    THING:  EVAL OF /EXP/ AND /BOX/
4Ø MAKE
    NAME:  "ERROR"
    THING:  RANDOMCHOOSE OF "Ø Ø Ø Ø 1 -2 4 -5 1Ø -15"
5Ø PRINT S S S S S S S  (S is the LOGO abbreviation for SENTENCE)
   "SOMEONE SAYS THAT THE RESULT OF EVALUATING"
   /EXP/ "WITH" SECOND OF /EXP/ "=" /BOX/ "IS"
   SUM OF /RESULT/ AND /ERROR/
6Ø PRINT "TYPE YES IF HE'S RIGHT AND NO IF HE'S WRONG."
7Ø MAKE
    NAME:  "ANSWER"
    THING:  REQUEST
8Ø TEST BOTH (IS /ANSWER/ "YES") AND (IS /ERROR/ "Ø")
9Ø IF TRUE PRINT "GOOD.  HERE'S THE NEXT PROBLEM"
1ØØ IF TRUE EVAL-CHECK
11Ø TEST IS /ERROR/ "Ø"
12Ø IF FALSE PRINT SENTENCE OF "THE RIGHT ANSWER IS"
    /RESULT/
13Ø PRINT "TRY ANOTHER PROBLEM"
14Ø EVAL-CHECK
END
```

RANDOMCHOOSE is a procedure for randomly choosing one of the elements in its input list. Thus, in line 4Ø RANDOMCHOOSE will either choose Ø (about 4 times out of 1Ø in the long run) or one of the positive or negative numbers listed. The number chosen is used to modify the actual result of evaluation as an additive error. The effect of EVAL-CHECK is illustrated as follows.

```
←EVAL-CHECK
SOMEONE SAYS THAT THE RESULT OF EVALUATING 24 BB + 2 WITH
BB = 3 IS 78
TYPE YES IF HE'S RIGHT AND NO IF HE'S WRONG
*YES
THE RIGHT ANSWER IS 74
TRY ANOTHER PROBLEM
  : : :
```

4.2 Generating Equations and Testing Solutions

We can modify EVAL-DRILL and EVAL-CHECK so that, instead of testing expression evaluation, they test equation solving. To start with, we can generate random equations instead of random expressions.

```
TO RANDOMEQ
1Ø MAKE
    NAME:   "TIMESNUM"
    THING:   RANDOMNUMBER OF 99
2Ø MAKE
    NAME:   "SUMNUM"
    THING:   RANDOMNUMBER OF 99
3Ø MAKE
    NAME:   "RIGHTNUM"
    THING:   SUM OF (PRODUCT OF /TIMESNUM/
             AND RANDOM) AND /SUMNUM/
4Ø OUTPUT S S S S
   /TIMESNUM/ "/BOX/ +" /SUMNUM/ "=" /RIGHTNUM/
END
```

Two-digit random numbers are chosen independently for the product and sum elements, in lines 1Ø and 2Ø. A number /RIGHTNUM/ for the right side of the equation is then computed from the product and sum numbers just chosen. Instead, we could have simply chosen /RIGHTNUM/ as a random number. If we had done that, the resulting equation would not always have an integer solution, and we want to begin with relatively simple problems, where /BOX/ is a single digit integer. The resulting equation is composed in line 4Ø. Here is an example illustrating the effect of RANDOMEQ.

```
←PRINT RANDOMEQ
23 /BOX/ + 74 = 12Ø
←
```

Most beginning students of school algebra do not know how to solve equations, even those of such simple form, using a general method. So, putting ourselves in the same position,

let's assume for now that we do not know how to write a procedure SOLVE for solving equations like those generated by RANDOMEQ. Then, we are more limited with equations than we were in treating expressions where we had an evaluation procedure, EVALUATE. Nevertheless, given an equation, we can test whether a proposed solution is valid. So we can write procedures for equations, SOLVE-DRILL and SOLVE-CHECK, that are analogous in their overall effect to EVAL-DRILL and EVAL-CHECK.

```
TO SOLVE-DRILL
1Ø MAKE
    NAME:  "EQ"                    (Generate a random equation)
    THING:  RANDOMEQ
2Ø PRINT /EQ/                      (Print it)
3Ø PRINT "WHAT IS /BOX/?"          (Ask user for a solution)
4Ø MAKE
    NAME:  "ANSWER"                (Get his answer)
    THING:  REQUEST
5Ø MAKE
    NAME:  "LEFT EXP"              (Get left side of equation)
    THING:  BUTLAST OF BUTLAST OF /EQ/
6Ø MAKE
    NAME:  "RIGHT NUMBER"          (Get right side - a number)
    THING:  LAST OF /EQ/
7Ø MAKE
    NAME:  "LEFT NUMBER"           (Evaluate left side with
    THING:  EVAL OF /LEFT EXP/     user's answer)
            AND /ANSWER/
8Ø TEST IS /LEFT NUMBER/ /RIGHT NUMBER/  (Is the result correct?)
9Ø IF TRUE PRINT "RIGHT!"
1ØØ IF FALSE PRINT S S S S S S
    "WRONG.  IF /BOX/ IS"
    /ANSWER/
    /LEFT EXP/
    "IS"
    /LEFT NUMBER/
    "NOT"
    /RIGHT NUMBER/
11Ø SOLVE-DRILL                    (Present another problem)
END
```

Here is an illustration of the use of SOLVE-DRILL.

```
←SOLVE-DRILL
12 /BOX/ + 19 = 55
WHAT IS /BOX/?
*2
WRONG.  IF /BOX/ IS 2, 12 /BOX/ + 19 IS 43 NOT 55
7 /BOX/ + 11 = 6Ø
WHAT IS /BOX/?
*7
RIGHT!
89 /BOX/ + 27 = 116
WHAT IS /BOX/?
  : : :
  : : :
```

A number of extensions and embellishments can be made to SOLVE-DRILL. For example, we can incorporate negative integers, larger numbers, non-integer solutions, a progression of problems of increasing complexity, and so on. As an illustration, let's time the user's response by using the CLOCK operation in LOGO. The output of CLOCK is the elapsed time in seconds since the computer's built-in second-clock was last reset. All we need do to modify SOLVE-DRILL is add the following instructions:

```
5 RESET CLOCK                          (This sets the clock to zero)

45 MAKE
    NAME:  "RESPONSE TIME"
    THING:  CLOCK

1Ø2 TEST GREATERP OF /RESPONSE TIME/
    AND 15                             (Does response time exceed 15
                                        seconds?)
1Ø4 IF TRUE PRINT SENTENCE OF SENTENCE OF
    "YOU TOOK" /RESPONSE TIME/
    "SECONDS TO ANSWER"
1Ø6 IF FALSE PRINT SENTENCE OF SENTENCE OF
    "THAT WAS FAST!  YOU ONLY NEEDED"
    /RESPONSE TIME/ "SECONDS TO ANSWER"
```

The effect of these changes in SOLVE-DRILL is evident, as in the following.

```
←SOLVE-DRILL
2 /BOX/ + 15 = 25
WHAT IS /BOX/?
↑5
RIGHT!
THAT WAS FAST!  YOU ONLY NEEDED 7
SECONDS TO ANSWER
  : : :
```

We can easily write a procedure SOLVE-CHECK, paralleling EVAL-CHECK, for checking whether a given number is a valid solution to a given equation. Instead of writing another procedure that artificially manufactures the solution to an equation, though, let's consider how we might develop a procedure for explicitly solving equations like those we have been generating and testing.

4.3 Solving Equations

We might have started with a less-sophisticated SOLVE-DRILL quiz which simply informed a user "WRONG!" when he gave an incorrect solution to the equation. We went a step further by showing the user the result of evaluating the left side of the equation with his false solution. If we wished, we could have told him the correct solution (since we had prescribed it in advance when we generated the equation with RANDOMEQ). Even better, we might like to tell him *how* to get the right solution.

If "we", however, are beginning students who do not ourselves know how to solve the equation, we evidently cannot so inform the user. The teacher can adopt at least three courses of action with students to help them acquire the solution algorithm.
(1) He can give *them* the problem, by asking them to modify SOLVE-DRILL so that it will tell the user how to solve the equation if the user's solution is wrong. Even though the students do not know how to do this, the teacher can tell them -

"Try!". In fact, this method has been effective in classroom work with very determined students.

(2) He might try to lead them to the algorithm by giving them, for example, a graduated series of problems in which one coefficient is fixed while the other is systematically varied, and ask them to look for patterns. For some students this will be helpful; others will see the appropriate patterns, but not the algorithm!

(3) He might *tell* them the algorithm and ask them to verify it.

Before settling on any or all of these, we'll try another idea -- we'll turn the equation-solving problem into a black box function problem by incorporating the solution algorithm into a secret (non-listable) procedure, say SPHINX. Students will use SPHINX and try to fathom its operation -- and we will make it easy for them to do so.

```
TO SPHINX
1Ø PRINT "GIVE ME AN EQUATION AND I'LL SOLVE IT"
2Ø MAKE
    NAME:  "EQN"
    THING:  REQUEST
3Ø PRINT "THE SOLUTION TO YOUR EQUATION IS GIVEN BY"
4Ø PRINT S S S S S S
   SECOND OF /EQN/
   "= ("
   LAST OF /EQN/
   "-"
   LAST OF BUTLAST OF BUTLAST OF /EQN/
   ") /"
   FIRST OF /EQN/
5Ø MAKE
    NAME:  "RESULT"
    THING:  DIVISION OF (DIFFERENCE OF LAST /EQN/
            AND LAST BUTLAST BUTLAST /EQN/) AND
            (FIRST /EQN/)
6Ø TEST IS LAST OF /RESULT/ "Ø"        (Is the remainder zero?)
```

```
7Ø IF TRUE PRINT S S S
    "THE NUMERICAL SOLUTION IS"
    SECOND OF /EQN/
    "="
    FIRST OF /RESULT/
8Ø IF FALSE TYPE S S S S
    "THE NUMERICAL SOLUTION IS"
    SECOND OF /EQN/
    "="
    FIRST OF /RESULT/
    WORD OF WORD OF (LAST OF /RESULT/)
                    "/"
                    FIRST OF /EQN/
END
```

In Line 5Ø, the result of the LOGO division operation is a two-word sentence. The first word is the quotient; the second word is the remainder. The student might use SPHINX as follows.

```
←SPHINX
GIVE ME AN EQUATION AND I'LL SOLVE IT
*3 X + 7 = 22
THE SOLUTION TO YOUR EQUATION IS GIVEN BY
X = ( 22 - 7 ) / 3
THE NUMERICAL SOLUTION IS X = 5

←SPHINX
GIVE ME AN EQUATION AND I'LL SOLVE IT
*4 X + 7 = 22
THE SOLUTION TO YOUR EQUATION IS GIVEN BY
X = ( 22 - 7 ) / 4
THE NUMERICAL SOLUTION IS X = 4 3/4
←
```

The relation between an original equation, say 4 x + 7 = 22, and the equation for its solution, x = (22 - 7) / 4 has been shown transparently by the use of SPHINX. We should be able to elucidate the relation between these two forms by the following kind of reasoning. The process of solving the given equation (let's call this *undoing* the equation) should be just the opposite one originally used in generating (i.e., *doing*) the equation.

For instance, consider the equation 4 x + 7 = 22. How was that built up starting with x? Here in effect is the ordered sequence of operations involved in building the equation.

(1) Take x
(2) Multiply it by 4
(3) Add 7
(4) Make the result 22

In order to solve this equation for x, that is to undo it, we must perform the opposite operations in the reverse order.

So our undoing sequence will be formed from the above operations taken in the order (4), (3), (2), (1). Also, wherever we added we'll subtract; wherever we multiplied we'll divide; we'll replace "make" by "take" and "take" by "make". Here then is the series of operations for undoing the equation.

(1) Take 22
(2) Subtract 7 (The result so far is 15)
(3) Divide it by 4 (The result is 3 3/4)
(4) Make the result x

Let's show that this algorithm, roughly stated: -- undoing an equation means performing the inverse operations in the opposite order -- has a precise formulation in the context of finding the inverse of a function.

4.4 Inverting Functions

Equations written in prefix notation look like functional expressions. Consider, for example, the infix equation:

3 /X/ + 5 = 17

In LOGO prefix notation, the left-side of this equation is written as the expression

SUM OF (PRODUCT OF 3 AND /X/) AND 5
(where the OF's and AND's are optional).

In this form, the expression can be regarded as a function defined by a procedure, say F OF /X/:

```
TO F /X/
1 OUTPUT SUM OF (PRODUCT OF 3 AND /X/) AND 5
END
```

The infix equation can be reinterpreted as a functional equation - the output of F OF /X/ is 17; the problem of solving this equation reduces to the problem of finding the inverse function of F, call it UNF OF /X/, for which UNF OF (F OF /X/) = /X/; thus, the solution to the equation will be found as the output of UNF OF 17.

So all we need do to solve the equation is to find UNF, the inverse of F. To help in doing this, let's redefine F in an equivalent but somewhat different form regarding it as the composition of the two functions, PRODUCT3 AND SUM5, defined as follows:

```
TO PRODUCT3 /X/
1 OUTPUT PRODUCT OF 3 AND /X/
END

←PRINT PRODUCT3 OF 9
27
←

TO SUM5 /X/
1 OUTPUT SUM OF /X/ AND 5
END

←PRINT SUM5 OF 27
32
←
```

```
TO F /X/
1 OUTPUT SUM5 OF PRODUCT3 OF /X/
END

←PRINT F OF 9
32
←
```

F /X/ is now the composition of the two single-input functions, SUM5 AND PRODUCT3. Thus, to "do" (i.e., perform) F /X/, we first perform the operation PRODUCT3 on /X/, and then perform the operation SUM5 on the result. Following the discussion at the end of Section 4.3, we make the conjecture that to "undo" F /X/, we must perform the inverse operations of SUM5 and PRODUCT3, and in the opposite order. That is,

```
TO UNF /X/
1 OUTPUT UNPRODUCT3 OF UNSUM5 OF /X/
END
```

where the inverse of SUM5 is UNSUM5 and the inverse of PRODUCT3 is UNPRODUCT3. These two inverses are easily identified functions:

```
TO UNSUM5 /X/
1 OUTPUT DIFFERENCE OF /X/ AND 5
END

TO UNPRODUCT3 /X/
1 MAKE
    NAME:  "D"
    THING:  DIVISION OF /X/ AND 3
2 TEST IS (LAST OF /D/) Ø              (Is the remainder Ø)
3 IF TRUE OUTPUT (FIRST OF /D/)        (Integer quotient)
4 OUTPUT SENTENCE OF
    FIRST OF /D/                       (Fractional remainder)
    WORD OF WORD OF LAST OF /D/
    AND "/" AND "3"
END
```

The effect of these inverse functions (for subtraction of 5 and division by 3) is evident:

```
←PRINT UNSUM5 OF 28
23
←PRINT UNPRODUCT3 OF 23
7 2/3
←PRINT UNPRODUCT3 OF 21
7
```

Moreover, UNSUM5 undoes SUM5 and UNPRODUCT3 undoes PRODUCT3:

```
←PRINT UNSUM5 OF SUM5 OF 123456789
123456789
←PRINT UNPRODUCT3 OF PRODUCT3 OF 123456789
123456789
```

Now let's try UNF on our functional equation problem, F OF /X/ = 17 to see if it gives a solution for /X/:

```
←PRINT UNF OF 17
4
```

To confirm this, let's see if /X/ = 4 is actually a solution:

```
←PRINT F OF 4
17
```

So UNF works in this case. And, when we go to the original problem-solving for /X/ in the equation 3 /X/ + 5 = 17, we find by substituting the result we just obtained (/X/ = 4) that it works there too.

Moreover, the problem we had in Section 3.4 -- finding the inverse of a composition of functions -- seems to yield to the same "undoing" procedure. In particular, consider the illustrative problem used there: REV OF LAG OF DUB OF /N/ = 2Ø864Ø864. Our undoing procedure gives for the inverse of the composite function REV OF LAG OF DUB, the composite function UNDUB OF UNLAG OF UNREV. (See Section 3.4 for the definitions of the three component functions UNREV, UNLAG, and UNDUB.)

So let's apply this inverse, using the number 2Ø864Ø864 as an input.

```
←PRINT UNDUB OF UNLAG OF UNREV OF 2Ø864Ø864
2468Ø
←
```

We can see that this must be so because,
(1) UNREV OF 2Ø864Ø864 is 468Ø2468Ø2
(2) UNLAG OF 468Ø2468Ø2 is 2468Ø2468Ø
(3) UNDUB OF 2468Ø2468Ø is 2468Ø.
So, /N/ = 2468Ø should be a solution to the given problem. Let's confirm that:

```
←PRINT REV OF LAG OF DUB OF 2468Ø
2Ø864Ø864
←
```

Given a chain of functions each of which has an inverse, we can easily write the undoing procedure (i.e., for generating the chain of the inverse functions composed in the opposite order):

```
TO UNDO /CHAIN/
1Ø TEST IS /CHAIN/ /EMPTY/          (When chain is exhausted)
2Ø IF TRUE OUTPUT /EMPTY/           (terminate)
3Ø TEST IS (LAST OF /CHAIN/) "OF"
4Ø IF TRUE OUTPUT UNDO OF           (Ignore "OF" terms)
    (BUTLAST OF /CHAIN/)
5Ø OUTPUT SENTENCE OF                  (Add name of inverse of last
   WORD OF "UN" AND (LAST OF /CHAIN/) function in chain to front of
   UNDO OF (BUTLAST OF /CHAIN/)        inverse chain and repeat with
END                                    rest of chain)
```

```
←PRINT UNDO OF "F OF G OF H OF I"
UNI UNH UNG UNF
←PRINT UNDO OF "REV LAG DUB"
UNDUB UNLAG UNREV
```

Using UNDO, then, we can write a procedure for solving equations whose left sides are function chains and whose right sides are numbers.

```
TO FUNDO /CHAIN/ /NUMBER/
1Ø DO SENTENCE OF SENTENCE OF
    "OUTPUT"
    UNDO /EQUATION/
    /NUMBER/
END

←PRINT FUNDO "DUB REV DUB" 25252525
52
←PRINT FUNDO "SUM5 PRODUCT3" 17
4
←PRINT FUNDO "REV SUM5 LAG PRODUCT3 DUB" 61121
37
```

If we develop the appropriate procedures for converting infix equations to functional equations, we can use FUNDO to solve many kinds of equations in algebra, of variable as well as fixed forms. But all of this depends on the use of the undoing algorithm, and although we have pragmatic evidence that supports it, we have not established its validity, nor even its plausibility.

We can show its plausibility by making its operation somewhat more transparent. Consider a chain of functions, F, given by F1 OF F2 ... OF F(N-1) OF FN. According to the undoing procedure the inverse of the chain, call it G, is UNFN OF UNF(N-1) ... OF UNF2 OF UNF1 and for any /X/ in the domain of F, the equation F OF /X/ = /NUMBER/ has the solution /X/ = G OF /NUMBER/. Therefore (by replacing /X/ in the equation), F OF G OF /NUMBER/ = /NUMBER/ by writing out F and G as chains, the expression F OF G expands to (F1 OF F2 ... OF F(N-1) OF FN) OF (UNFN OF UNF(N-1) ... OF UNF2 OF UNF1) and this can be written,
F1 F2 ... F(N-1) (FN UNFN) UNF(N-1) ... UNF2 UNF1. But, since UNFN is the inverse of FN, FN OF UNFN outputs its input, and since it does not change its input, can be deleted with no effect. This leaves F1 F2 ... (F(N-1) UNF(N-1)) ... UNF2 UNF1. By the same argument F(N-1) OF UNF(N-1) can be deleted, then so can the

successive couples F(N-2) UNF(N-2), ... F2 UNF2, through F1 UNF1. Finally, then F OF G reduces to the identity function, SAME, which leaves its input unchanged. That is, F OF G OF /NUMBER/ = /NUMBER/, as asserted, and G is, in fact, an inverse for F.

This plausibility argument can be made rigorous in this context. Perhaps it is easier to do so (for beginning students of algebra) by going back to the representation of functions as lists of ordered pairs. In this, our original context, we can establish the undoing algorithm by exhaustive enumeration. We begin by developing a procedure for evaluating a chain composed of N pair lists with a given input, FIND-OUTPUT /INPUT/ /PAIR LIST 1/ ... /PAIR LIST N/. We prove that the procedure is valid by induction, using the procedures MATCH and COMPOSE.

Then, we use the same ideas going in the reverse direction to develop the inverse procedure FIND-INPUT for solving a chain of N pair lists for the input whose evaluation leads to a given output: FIND-INPUT /PAIR LIST 1/ ... /PAIR LIST N/ /OUTPUT/. Again, we develop an inductive argument this time using the pair list inversion procedure SWITCH-PAIRS, along with MATCH and COMPOSE.

4.5 Harder Equations

Thus far we have been discussing equations of a limited kind where the right side is a number, not an expression in /X/. We have spent a great deal of time on this special case to make the notion of a solution algorithm apparent and unmysterious to beginning students. So often this "simple" idea is not grasped and a conceptual barrier to further understanding of algebra almost certainly results.

In continuing work we shall extend the discussion beyond equations like 7 * /X/ + 3 = 12 and study those like 7 * /X/ + 3 = 4 * /X/ + 12 with linear expressions on both sides of =. We shall regard the expression on each side of the equation as a procedure.

After writing procedures for manipulating expressions, and studying the different effects of various operations, we shall consider how to manipulate operations involving two expressions. In particular, we shall first look for operations which preserve the equality relations and then for sequences of operations which define solution algorithms.

LOGO: Functions and Equations
Problems

Introduction

The problems here go along with the expository material in the unit LOGO: Functions and Equations, Teacher's Text. The numbering used here is the same as that used in the main text.

Problems

Section 2

1. Usually, for the sake of clarity, the pair of elements a and b is written as (a,b). Thus, for the list given on page 3 we would have

"(A,B) (B,C) (C,D) ... (Z,A)"

(A) Write a procedure SEPARATE which translates a list of pairs, given in the form of a LOGO sentence into this form.

(B) Write a procedure for going the other way, UNSEPARATE. This procedure translates a list of pairs, written as above with commas and parentheses, into LOGO sentence form.

2. Write a procedure TABLE which takes a LOGO pair list and prints the data as a table - for example, TABLE "A B B C C D" gives the printed result:

```
A  B
B  C
C  D
```

3. Taking all elements from the set of single digit integers, write down the following relations as pair lists:

(A) All pairs such that the second element is three times the first element.

(B) The second element is the square of the first.

(C) The first is the square of the second.

(D) Twice the first plus the second is 9.

(E) The difference between the second and the first is 1.

(F) The second is divisible by the first without remainder.

4. (A) Write a procedure COMPARE-MATCHES /ELEMENT/ /PAIR LIST 1/ /PAIR LIST 2/ to find whether or not a given element has the same matching elements on two different pair lists.

(B) Using COMPARE-MATCHES, write a procedure COMPARE-LISTS /PAIR LIST 1/ /PAIR LIST 2/ to decide whether or not two pair lists are equivalent in the sense that any first element from either list has the same set of matched second elements in both lists.

(C) Consider the following shorter procedure for comparing two pair lists:

```
TO COMPARE-LISTS-FAST /PAIR LIST 1/ /PAIR LIST 2/
1Ø TEST IS /PAIR LIST 1/ /PAIR LIST 2/
2Ø IF TRUE PRINT "TRUE"
3Ø PRINT "FALSE"
END
```

Does this procedure always work? If not, give an example where this procedure does not give the same result as the procedure you wrote above for part (B).

Section 2.1

1. How would you find the range of a given relation if you were allowed only the use of DOMAIN and SWITCH-PAIRS?

2. If two pair lists /A/ and /B/ are equivalent in the sense defined above (Section 2, problem 4), when are the associated lists, SWITCH-PAIRS OF /A/ and SWITCH-PAIRS OF /B/ also equivalent?

Section 2.2

1. Extend the plotting procedure PLOT to domains and ranges which need not be single characters.

Note: This task is most easily done by splitting it into two parts: the printing of the domain elements can be done by a new procedure DOMAINLIST, say, which types the first character of each domain element on one line, the second of each on the next, and so on till all characters of the longest element are printed. The problem of listing range elements can be solved by padding out the shorter words with spaces so that all range elements are of the same length.

2. Demonstrate the truth of the statement given on page 13 that there is no way of reordering /PAIR LIST/ to get the same result as PRETTYPLOT.

3. What function would give the graphs indicated below? First specify the function as a set of pairs and then as a LOGO procedure.

(A)

```
  Ø12345
Ø .
1  .
2   .
3    .
4     .
5      .
```

(B)

```
  Ø12345
Ø      .
1     .
2    .
3   .
4  .
5 .
```

Section 2.3

1. The definition of a function in terms of pairs is that every element in the domain has exactly one element in the range associated with it. How can this be stated in terms of graphs? In other words, what restriction on a graph assures us that it is the graph of a function?

2. Often we have relations in which the elements in the domain are themselves pairs. For example, each element in the domain could be a pair of numbers, specifying a point on the floor and the corresponding element in the range the temperature at that point. If the elements in the domain are written as LOGO words (with commas instead of spaces between pairs) then all the procedures we have written still work for these relations.

(A) Multiplication can be considered as a relation which takes a pair of elements into their product. Write some pairs defining this relation. Write a procedure which generates all pairs whose domain elements consist of pairs of single-digit numbers. How is this list related to multiplication tables?

(B) Use the list gotten in Part (A) as part of a procedure for multiplying any two single-digit numbers.

(C) Plot the relation defined by that list.

(D) Another way of plotting a relation, whose domain consists of pairs, is to write the element of range itself (instead of a ".") at the point in the plot defined by the first pair. With the help of the existing PLOT procedure, write a set of procedures which does this.

(E) Note that it is difficult to use these procedures if elements of the range are larger than one character. Why is it difficult to use this sort of plotting for relations other than functions? Can you think of any ways to overcome this difficulty?

Section 2.4

1. (A) Demonstrate the fact that decoding a message is the same as encoding using SWITCH-PAIRS of the encoding list.

(B) Use this fact to write a simpler version of DECODE than the one given in the text.

2. An interesting method of coding is to leave the letters making up each word the same, but to change their positions. Reversing each word is a simple example of this type of coding. More generally, we can describe such an encoding by giving a list of "sequencing" numbers such as 4 3 1 2 6 5. This means that we put the fourth letter of the word first, the third letter second, etc. in encoding each word. If the word does not contain four letters, for example, then the fourth character is taken to be empty. Thus, encoding of CAT using the above list is TAC and that of TURTLE is TRTUEL.

(A) Write a procedure with two inputs -- the "sequencing list" and the message -- to perform this sort of coding. Don't forget to allow for the case when a word in the message has more characters than does the encoding list.

(B) Write a corresponding procedure for decoding messages.

3. Let us say that a message is coded "doubly" -- first using one encoding and then encoding the result with another encoding.

(A) How would you retrieve the original message?

(B) Some encodings are also decodings in the sense that applying them twice gives the original message. Reverse is an example of this type of code. Invent some other examples.

Section 3.1

1. (A) Let us say we have a function consisting of a list of pairs of numbers. If we added 1 to the second element of each pair, would the resulting relation still be a function? How is the graph of this new relation related to the graph of the original one?

(B) Answer the same questions for the case where 1 is added to the first element of each pair.

(C) How would you make the same changes on a function defined by a procedure?

2. (A) Given a five element domain and a five element range, how many different functions can be made?

(B) Write a procedure ALLFUNCTIONS /DOMAIN/ /RANGE/ which generates all possible functions with domain /DOMAIN/ and range /RANGE/.

3. (A) Write a procedure CONSTANTP which takes a list of pairs as an input and outputs true or false as the relation described is or is not a constant function.

(B) Write IDENTITYP which checks whether a list describes the identity function.

(C) Write DOUBLEP which checks that each element in the range is twice the corresponding domain element.

(D) Write MULTIP which checks that each element in the range is the same multiple of the corresponding domain element.

(E) Write LINEARP which checks a list of pairs for the form A times the domain element plus B where A and B are the same numbers for all pairs on the list.

Section 3.2

1. Use the LOGO DO command to help write a black box linear function generator, MAKE-A-BOX /A/ /B/ /BOXNAME/. MAKE-A-BOX creates a procedure named /BOXNAME/ with the input /X/ and the output SUM OF PRODUCT OF /A/ /X/ AND /B/.

2. Black boxes can be constructed using pair lists.

(A) Write a procedure which uses the procedure MATCH to construct a black box from a given pair list.

(B) Using PICTURELIST, construct such a black box by typing in a graph.

3. Thus far we have worked with black boxes whose outputs are numbers, words, or other strings. We can construct black boxes which have graphical outputs instead.

(A) Make a black box which has two inputs, each a row of X's and outputs the rectangle having those numbers of X's as the lengths of its sides.

(B) Make a black box which, given a row of X's, outputs a triangle, with base having that number of X's and 1 less in each subsequent row. Thus,
BLACK-BOX "XXXX" gives
XXXX
XXX
XX
X

4. Often it is useful to have plots of portions of the graph of a black box function defined by a LOGO procedure. Write a procedure which takes as inputs the name of such a procedure and the domain of interest, and generates the corresponding plot.

5. Sometimes the order in which black boxes are composed does not effect the output. (Usually, though, this is not the case.)

(A) Write a procedure ORDERP /BLACK-BOX 1/ /BLACK-BOX 2/ /INPUTS/ which tests whether the order of composition of the two specified black boxes is different for the specified list of inputs.

(B) Try the procedure on the following pairs: SUM /INPUT/ AND 3; SUM /INPUT/ AND 5.

(C) SUM (DOUBLE /INPUT/) AND 3; SUM (DOUBLE /INPUT/) AND 5.

(D) DOUBLE OF /INPUT/; TRIPLE OF /INPUT/.

(E) BUTFIRST OF BUTLAST OF /INPUT/; BUTLAST OF BUTFIRST OF /INPUT/.

6. Let us consider inputting successive integers to a black box. The black box is said to be periodic if the sequence of outputs repeats itself after some point. A simple example is the black box which outputs the remainder of its input divided by 4. If we successively input 1 2 3 4 5 6 7 8 9 10, we get out the sequence 1 2 3 1 2 3 1 2 3 1 etc.

(A) Write a procedure which tests whether or not a given black box is periodic.

(B) Will your procedure work for all periodic functions? If not, what are the constraints?

Sections 4 - 4.5

1. (A) Does the "built-in" LOGO function FIRST have an inverse? That is, if FIRST OF /W/ is /X/, can you write a procedure defining the function UNFIRST, where UNFIRST OF /X/ is /W/?

(B) What about the LOGO functions BUTFIRST, LAST, BUTLAST, and COUNT -- can you write procedures defining the inverse functions UNBUTFIRST, UNLAST, UNBUTLAST, or UNCOUNT?

2. Can you solve any of the following equations for the input /X/?

(A) BUTFIRST OF /X/ = "ULP"
(B) FIRST OF /X/ = "WHERE"
(C) COUNT OF /X/ = 2
(D) BUTFIRST OF BUTLAST OF /X/ = "IDDL"

3. How many solutions can you find for the following sets of simultaneous LOGO equations?

(A) BUTFIRST OF /X/ = "ULP"
BUTLAST OF /X/ = "GUL"

(B) LAST OF /X/ = "O"
FIRST OF /X/ = "N"
COUNT OF /X/ = 2

4. Consider the two-input LOGO function WORD.

(A) Can you write a LOGO procedure for the inverse function UNWORD so that if WORD OF /U/ AND /V/ is /W/, then UNWORD OF /W/ gives back /U/ and /V/?
(For example, WORD OF "GL" and "UB" is "GLUB" so UNWORD OF "GLUB" would output the sentence "GL UB")

(B) If you are unable to write a function UNWORD as specified in part (A), describe your difficulty and illustrate it with specific examples.

(C) Can you write procedures for left and right inverse functions UNWORDL and UNWORDR for the function WORD, so that:
UNWORDL /V/ AND /W/ outputs /U/, and UNWORDR /U/ and /W/ outputs /V/?

5. Solve for /X/ in the following LOGO equations:
(A) UNWORDL "ZAG" AND /X/ = "ZIG"
(B) UNWORDR "LEFT" AND /X/ = "RIGHT"
(C) WORD OF /X/ AND "ME" = "SOME"

6. Write a program which takes as its inputs two black-box function names and a range of numbers. It is to output the numbers within the given range for which the two black boxes give the same output.

7. (A) Build a black box which outputs the sum of twice its input and 3. Build another one which outputs its input.

(B) Use the program written in problem 6 to find those inputs for which the outputs of the two black boxes are the same.

(C) How does the program written in part (B) relate to "solving" the equation $2x + 3 = x$?